Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

Sharper Crypto-API Analysis

Entwicklung eines integrierten Plugins zur statischen Code-Analyse der Nutzung von Krypto-APIs in C#

Masterarbeit

im Studiengang Medieninformatik
zur Erlangung des akademischen Grades
Master of Science

Fachbereich Medien
Hochschule Düsseldorf

Sebastian Leuer
Matrikel-Nr.: [redacted]
Datum: Januar 2019

Erst- und Zweitprüfer

Prof. Dr.-Ing. Holger Schmidt
Prof. Dr. Manfred Wojciechowski

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Masterarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind vollständig zitiert. Diese Arbeit wurde weder in gleicher noch ähnlicher Form einem anderen Prüfungsamt vorgelegt oder veröffentlicht. Ich erkläre mich ausdrücklich damit einverstanden, dass diese Arbeit mittels eines Dienstes zur Erkennung von Plagiaten überprüft wird.

Ort, Datum *Sebastian Leuer*

Kontaktinformationen
Sebastian Leuer
[redacted]
—
[redacted]

Zusammenfassung

Sharper Crypto-API Analysis
Entwicklung eines integrierten Plugins zur statischen Code-Analyse der Nutzung von Krypto-APIs in C#

Sebastian Leuer

Untersuchungen haben ergeben, dass bis zu 95% aller verfügbaren Android Apps mindestens einen Programmierfehler bei der Verwendung von sogenannten Krypto-APIs besitzen. Bei korrekter Anwendung dieser APIs können mit ihnen sensible System- und Benutzerdaten vor Angriffen geschützt werden. Bisher gibt es nur wenige verfügbare Tool-basierte Hilfsmittel für Programmierer, welche die Verwendungsfehler identifizieren können. Für die Programmiersprache Java konnten bislang nur sehr wenige dieser spezialisierten Hilfsmittel gefunden werden, darunter das statische Codeanalysetool CogniCrypt. Ein derartiges Werkzeug für C# konnte nicht gefunden werden.

Diese Arbeit befasst sich mit der Entwicklung eines Plugins für die IDE Visual Studio zur statischen Codeanalyse namens *Sharper Crypto-API Analysis*. Sein Hauptziel ist, Programmierern beim Verwenden der .NET Krypto-API mit der Programmiersprache C# zu unterstützen. Durch Beobachtungen im Entwicklungsprozess sollen einerseits die Parallelen und Unterschiede zwischen der Java- und der C#-Welt im Umgang mit Krypto-APIs erkennbar werden. Andererseits beschreibt diese Arbeit den erforderlichen Aufwand, um ein hoch spezialisiertes, statisches Codeanalysetool zu entwickeln. Die Beobachtungen zeigen, dass die Unterschiede der Java- und C#-Technologien nicht oberflächlich, sondern hauptsächlich im Detail liegen und daher während der Implementierung des Tools besondere Aufmerksamkeit erfordern. Die Entwicklungsphase selbst zeigt sich als besonders anspruchsvoll, da mit Unsicherheitsfaktoren bezüglich der Funktions- und Verhaltensweisen von Visual Studio umgegangen werden müssen.

Das Hauptziel von Sharper Crypto-API Analysis konnte durch die Implementation von vier Hauptfunktionsgruppen umgesetzt werden. Die wichtigste von ihnen ist das Bereitstellen von Codeanalysen, welche ausschließlich auf die Verwendung der .NET Krypto-API spezialisiert sind. Die hier entwickelten Codeanalysen betrachten besonders Fehlerquellen, die Programmierer am häufigsten begehen. Um den Programmierern nachhaltig Wissen bezüglich der Krypto-API zu vermitteln, führt diese Arbeit erweiterte Analyseberichte ein, die zusätzlich über Schwachstellen oder Angriffsszenarien informieren. Besonderheiten und Alleinstellungsmerkmale der anderen Funktionsgruppen werden in dieser Arbeit unter besonderer Berücksichtigung von Implementationshinweisen dokumentiert.

Abstract

Sharper Crypto-API Analysis
Development of an integrated plugin for static code analysis when using cryptographic APIs in C#

Sebastian Leuer

Recent researches have shown that up to 95% of all available Andriod Apps contain at least one programming error caused by wrongly applying so-called Crypto-APIs. By using these APIs, applications can ensure protection of sensitive data against attacks. Currently there is a very limited amount of program-based tools available which help programmers identifying their mistakes in using Crypto-APIs. The programming language Java seems to be most prominent as many of the few tools found are specialized for this language. One of them is called CogniCrypt. A specialized tool for the C# programming language however could not be found.

The thesis at hand addresses the development of a static code analysis plugin for the IDE Visual Studio called *Sharper Crypto-API Analysis*. To assist programmers using the .NET Crypto-API in C# shall be its main goal. Observations during the development process shall point out differences between the Java and C# ecosystems in handling Crypto-APIs. Furthermore the work at hand describes the necessary expenditure to develop a static code analysis tool. On the one hand the observations show differences between Java and C#, which only become obvious after a careful examination. Hence, attention must be paid during the implmentation of the tool. On the other hand the development process revealed itsself to be challenging because of unpredictable and undocumented behaviour of the host platform Visual Studio.

The main goal of Sharper Crypto-API Analysis could be achieved by implementing four primary function classes. The most important class contains code analyzers which investigate exspecially the usage of the .NET Crypto-API. In particular the analysis specialized at errors which programmers tend to do most commonly. In addition to analyzing bare source code the tool also presents more detailed reports that shall help programmers to understand weaknesses and attacks their code allows to perform and most importantly how to fix the identified issues. Lastly this work will also document the other function classes and point out their features and behaviours with a special consideration on implementation details.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

AES Advanced Encryption Standard
API Application Programming Interface
CBC Cipher Block Chaining
CIL Common Intermediate Language
CLI Common Language Infrastructure
CLR Common Language Runtime
CVE Common Vulnerabilities and Exposures
CWE Common Weakness Enumeration
DES Data Encryption Standard
ECB Electronic Code Book
HMAC Keyed-Hash Message Authentication Code
IDE Integrierte Entwicklungsumgebung
IV Initialisierungsvektor
JCA Java Cryptographic Architecture
JIT Just-in-time
MD5 Message-Digest Algorithm 5
MEF Managed Extensibility Framework
MVVM Model View ViewModel
NIST National Institute of Standards and Technology
PBDKF Password-Based Key Derivation Function
SCAT Static Code Analysis Tool
SDK Software Development Kit
SDLC Software Development Life Cycle
SHA Secure Hash Algorithm
SSDLC Secure Software Development Life Cycle
TDES Triple Data Encryption Standard
WPF Windows Presentation Foundation

Kapitel 1

Einleitung

„Program testing can be used to show the presence of bugs, but never show their absence!“

Edsger Wybe Dijkstra
(∗1972)

Softwareprogramme und vor allem mobile Apps haben heutzutage einen wichtigen Anteil im Alltag des Menschen. Statistiken gehen davon aus, dass im Jahr 2017 weltweit über 167 Mrd. Apps von den beiden größten Plattformen Apple App Store und Google Play Store heruntergeladen wurden[1][2]. Allein im Apple App Store sind über 800.000 verschiedene Apps der Kategorien: Business, Lifestyle, Finanzen, Soziale Medien oder Shopping verfügbar (knapp 25% aller Apps)[3]. Diese immens hohe Vielfalt an verfügbaren Apps ist unter anderem damit zu erklären, dass die Entwicklung dieser Apps durch neue Technologien und Konzepte wie dem Cross-Platform Development Ansatz für die Entwickler immer einfacher wird (Heitkötter, Hanschke & Majchrzak, 2012, S. 14). Das besondere an den genannten App-Kategorien ist, dass viele der hier zugeordneten Apps mit hoher Wahrscheinlichkeit sensible Informationen ihrer Nutzer abfragen, die dann im weiteren Verlauf der Anwendung verarbeitet oder gespeichert werden. Um die Daten vor Angriffen wie Diebstahl oder Manipulation zu schützen, müssen die Entwickler Maßnahmen vorsehen, welche die Sicherheit der sensiblen Informationen gewährleisten.

Die Informationssicherheit befasst sich gemäß der ISO 27000 (ISO/IEC, 2018) mit der Erhaltung von den Schutzzielen Vertraulichkeit, Integrität und Verfügbarkeit, auch bekannt als CIA-Dreieck (Peltier, 2013). Der Begriff *Sicherheit* ist im Deutschen mehrdeutig und beinhaltet sowohl *Safety* (dt. Unfallvermeidung oder Funktionssicherheit) als auch *Security* (dt. Kriminalprävention). Safety im Sinne von Unfallvermeidung bedeutet, dass das IT-System Maßnahmen bereitstellt, seine Umwelt und die Menschen zu schützen[4]. Eckert beschreibt Safety im Sinne von Funktionssicher-

[1] Apple App Store: https://de.statista.com/statistik/daten/studie/382129/umfrage/anzahl-heruntergeladener-apps-fuer-iphone-und-ipad-weltweit-als-prognose/, Zugriff: 11.12.2018

[2] Google Play Store: https://de.statista.com/statistik/daten/studie/243412/umfrage/anzahl-von-downloads-im-google-play-store/

[3] https://www.pocketgamer.biz/metrics/app-store/categories/, Zugriff: 11.12.2018

[4] https://www.tuev-nord.de/explore/de/erklaert/was-ist-der-unterschied-zwischen-safety-und-security/, Zugriff: 11.12.2018

heit als Eigenschaft, in welcher das IT-System unter „allen (normalen) Betriebsbedingungen funktioniert“ (Eckert, 2013, S. 6). Ein IT-System ist unter anderem eine Anwendungssoftware, welche Informationen verarbeitet und speichert. Maßnahmen zur Verhinderung einer Kompromittierung der Informationen fallen unter den Bereich der Security. Eine Software, die diese Maßnahmen besitzt, wird auch als *sichere Software* bezeichnet (John Viega & McGraw, 2011).

Im weiteren Verlauf dieser Arbeit werden sämtliche Begriffe wie „Angriff“, „Verwundbarkeit“ oder „Bedrohung“ gemäß den Definitionen aus der ISO 27000 verwendet.

Eine dieser Maßnahmen, die Softwareentwickler anwenden können, ist das Verwenden von kryptographischen Verfahren (Spitz, Pramateftakis & Swoboda, 2011, S. 37). Damit nicht alle Entwickler diese Verfahren immer aufs Neue programmieren müssen, werden sie in sogenannten Krypto-Application Programming Interfaces (APIs) angeboten. Krypto-APIs sind Programmbausteine oder Programmschnittstellen, mit denen Anwendungsentwickler kryptographische Aufgaben, wie die Verschlüsselung von Informationen, in ihre Anwendungen integrieren können. Jedoch ist nicht nur die korrekte Implementation seitens des Herstellers der APIs erforderlich, sondern vor allem auch ihre fehlerfreie Verwendung durch den Anwendungsentwickler ist essentiell für sichere Software. Programmierfehler bei der Verwendung von Krypto-APIs können Schwachstellen und Verwundbarkeiten in der Software produzieren und stellen daher eine Bedrohung des IT-Systems dar. Aus dem folgenden Beispiel von 2017 wird diese Bedrohung deutlich und zeigt auf, dass sich Softwareentwickler nicht immer an die Vorgaben halten, welche durch die Krypto-APIs definiert werden.

Betrachtet wird die App *Tagebuch app mit Schloss* (engl. Diary with lock). Im Google Play Store ist sie in der App-Kategorie Lifestyle gelistet. Die App besaß im Jahr 2017 einen schwerwiegenden Programmierfehler. Sowohl der Initialisierungsvektor (IV) als auch der geheime Schlüssel für das symmetrische Verschlüsselungsverfahren Advanced Encryption Standard (AES) waren im Quelltext als `string` Variable hartkodiert. Mittels einer einfachen Dekompilierung der App konnten daher beide Verschlüsselungsparameter ausgelesen werden, wie die Abbildung 1.1[5] demonstriert. Dieses Beispiel zeigt einen starken Verstoß der Nutzung einer Krypto-API. Allein der Name des Typs `SecretKeySpec` verdeutlicht, dass der Verschlüsselungsschlüssel geheim sein muss. Durch seine konstante Kodierung und dem Auslesen des Quelltexts bleibt ihm diese wichtige Eigenschaft jedoch vorenthalten. Der IV wird benötigt, um den Verschlüsselungsvorgang mit dem Betriebsmodus Cipher Block Chaining (CBC) einzuleiten und um die Entschlüsselung durchzuführen (Ehrsam, Meyer, Smith & Tuchman, 1978). Er sollte daher öffentlich sein, jedoch nicht konstant, sondern zufällig und unvorhersehbar (Rogaway, 1995). Mit einem immer gleichen IV wird der Geheimtext, auch Ciphertext genannt, bei gleicher Eingabe der zu verschlüsselnden Nachricht immer identisch sein. Die verschlüsselte Information wäre somit angreifbar. Auch vorhersehbare IVs stellen eine Schwäche dar[6]. Sind einem Angreifer IV, Schlüssel und Betriebsmodus (in diesem Fall der CBC-Modus ohne Padding) bekannt, kann dieser jede verschlüsselte Information, in diesem Fall die persönlichen Gedanken und

[5] https://1337sec.blogspot.com/2017/10/auditing-writediarycom-cve-2017-15581.html, Zugriff: 08.12.2018

[6] Beispiel, welches den Angriff auf einen vorhersehbaren IV vorstellt: https://defuse.ca/cbcmodeiv.htm, Zugriff: 09.12.2018

```
private String SecretKey = "f8djD2kfdU3lwuRu";
private Cipher cipher;
private String iv = "aBcNvn71dkdjf8hf";
private IvParameterSpec ivspec = new IvParameterSpec(this.iv.getBytes());
private SecretKeySpec keyspec = new SecretKeySpec(this.SecretKey.getBytes(), "AES");
```

Abbildung 1.1: Hartkodierte Verschlüsselungsparameter im Quelltext der App „Tagebuch app mit Schloss“

Gefühle desjenigen, der einen Tagebucheintrag geschrieben hat, wieder lesbar machen. Die App *Tagebuch app mit Schloss* war daher gar nicht so sicher, wie ihr Name versprach. Diese Schwachstelle wurde unter der Common Vulnerabilities and Exposures (CVE)-Nummer 2017-15582 eingetragen[7] und ist in der aktuellen Version der App behoben.

Auf Communityseiten wie Codeproject.com finden sich ebenfalls viele Beispiele der inkorrekten Nutzung von KryptoAPIs. In einem Beitrag[8] von 2007 wird beispielsweise das schon damals veraltete Data Encryption Standard (DES)-Verfahren zur Verschlüsselung benutzt. In einem anderen Artikel[9] von 2014 wird eine passwortbasierte AES Verschlüsselung mit Salt vorgestellt. Auch hier wird der IV nicht zufällig generiert. Problematisch an dieser Stelle ist, dass diese beiden Artikel mit jeweils mindestens 4,5 von 5 möglichen Punkten bewertet wurden und somit suggerieren, es würde sich um Lösungen handeln, die risikofrei angewendet werden können.

Auch die Hersteller von Krypto-APIs zeigen in ihren Dokumentation und Beispielen Schwächen. Ein bekanntes Beispiel stammte von Microsoft. Auf der Seite support.microsoft.com bietet der Softwarehersteller viele Hilfestellungen, Dokumentationen und Tutorials für die Verwendung der eigenen Programmiersprachen und Softwarebibliotheken an. Eines dieser Tutorials behandelte auch die Ver- und Entschlüsselung von Dateien mit C#[10]. Dieser Artikel enthielt jedoch viele Fehler und falsche Behauptungen. Darunter die Empfehlung, das Benutzerpasswort als Schlüssel und IV zu benutzen oder DES als Verschlüsselungsverfahren zu wählen. Eine ausführlichere Analyse dieses Tutorials, liefert ein Artikel von Robert Parks[11].

Sind schon Tutorials zur Nutzung von Krypto-APIs fehlerhaft, jedoch einfach nachzubauen, ist das Risiko sehr hoch, dass Programmierer den schwachen Code in ihre Anwendungen übernehmen (Meng, Nagy, Yao, Zhuang & Argoty, 2018). Laut einer umfassenden Analyse von Egele et al. wurden über 11.000 aktuell verfügbare Android Apps auf Fehler bei der Benutzung von Krypto-APIs überprüft. Bei 88% der überprüften Apps konnte mindestens ein Fehler diagnostiziert werden (Egele, Brumley, Fratantonio & Kruegel, 2013). Neuere Untersuchungen von Android Apps zeigen sogar eine Fehlerquote von 95% (Krüger, Späth, Ali, Bodden & Mezini, 2018).

[7] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15582, Zugriff: 08.12.2018

[8] https://www.codeproject.com/Articles/19538/Encrypt-Decrypt-String-using-DES-in-C, Zugriff: 09.12.2018

[9] https://www.codeproject.com/Articles/769741/Csharp-AES-bits-Encryption-Library-with-Salt, Zugriff: 09.12.2018

[10] https://web.archive.org/web/20170327154501/https://support.microsoft.com/en-us/help/307010/how-to-encrypt-and-decrypt-a-file-by-using-visual-c, Zugriff: 09.12.2018 (bei deaktiviertem JavaScript)

[11] https://medium.com/@bob_parks1/how-not-to-encrypt-a-file-courtesy-of-microsoft-bfadf2b0273d, Zugriff, 09.12.2018

Zu den häufigsten Fehlern gehören laut Krüger et al. die Verwendung alter, unsicherer Hashingverfahren, wie Message-Digest Algorithm 5 (MD5) oder Secure Hash Algorithm (SHA)-1 und Verschlüsselungsalgorithmen wie DES. Auch wenn nicht jede der gefundenen Schwachstellen eine Verwundbarkeit darstellt, so kann hier doch von einem erheblichen Problem der Programmierer gesprochen werden, Krypto-APIs zuverlässig einzusetzen. Rechnet man diese Fehlerquote auf die Anzahl der 2017 heruntergeladenen Apps im Google Play Store (94 Mrd.) hoch, so kommt man auf das ungefähre Ergebnis von 89 Mrd. Downloads. Das heißt auf jeden Menschen kommen in etwa 11,7 potenziell sicherheitskritische Android Apps[12].

Diese Arbeit setzt sich daher zum Ziel, den Softwareentwicklern zu helfen, Fehler in ihrem Quellcode eigenständig zu finden und zu beheben, bevor die Anwendung veröffentlicht wird und somit eine Bedrohung gegen die sensiblen Informationen ihrer Nutzer darstellt.

1.1 Motivation zur Entwicklung eines statischen Codeanalyseprogramms für C#

Die hier aufgeführten Beispiele zeigen deutlich, dass nicht nur die Anwendungsentwickler Schwierigkeiten haben Krypto-APIs zu verwenden, sondern zum Teil auch fehlerhafte Tutorials oder Dokumentationen der Hersteller dieser APIs existieren. Programmierer mit wenig Fachkenntnissen über Kryptographie sind daher auf eine Dokumentation mit korrekten Beispielen angewiesen. Microsoft hat daher entsprechend reagiert und den angesprochenen Artikel von ihrer Seite genommen. Ebenso wurden die Dokumentationen zu DES mit deutlich sichtbaren Informationen versehen, die auf neuere, sichere Verfahren hinweisen (s. Abbildung 1.3[13]). Weitere dieser Hinweise finden sich beispielsweise auch bei dem Hashingverfahren MD5[14]. Ein genaueres durchforsten der API-Dokumentationen zeigt jedoch, dass in einigen Codebeispielen noch immer das DES oder Triple DES Verschlüsselungsverfahren verwendet wird[15].

Ein weiterer Indikator, dass Entwickler zunehmend Schwierigkeiten haben Krypto-APIs zu benutzten, lässt sich anhand einer Analyse der häufig gebrauchten Fragewörter auf Stackoverflow.com feststellen. Wie in Abbildung 1.2[16] zu erkennen, stiegen die Fragen bezüglich APIs in den letzten Jahren stark an. Daraus kann abgeleitet werden, dass der Hilfsbedarf der Entwickler aktuell steigt und nicht, wie man möglicherweise annehmen könnte, sinkt. Ein weiterer Anhaltspunkt für die Aktualität des Themas ist, dass ab dem Jahr 2015 vermehrt wissenschaftliche Arbeiten veröffentlicht wurden, die den Umgang mit Krypto-APIs und deren Gebrauchsschwierigkeiten untersuchen. Abschnitt 3.2 befasst sich intensiv mit diesen Forschungen.

[12] Bei einer Weltbevölkerung von 7,6 Mrd. Menschen.

[13] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.des, Zugriff: 09.12.2018

[14] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.md5, Zugriff: 09.12.2018

[15] https://docs.microsoft.com/de-de/dotnet/api/system.security.cryptography.rfc2898derivebytes, Zugriff: 17.12.2018

[16] https://insights.stackoverflow.com/trends?tags=api, Zugriff: 08.12.2018

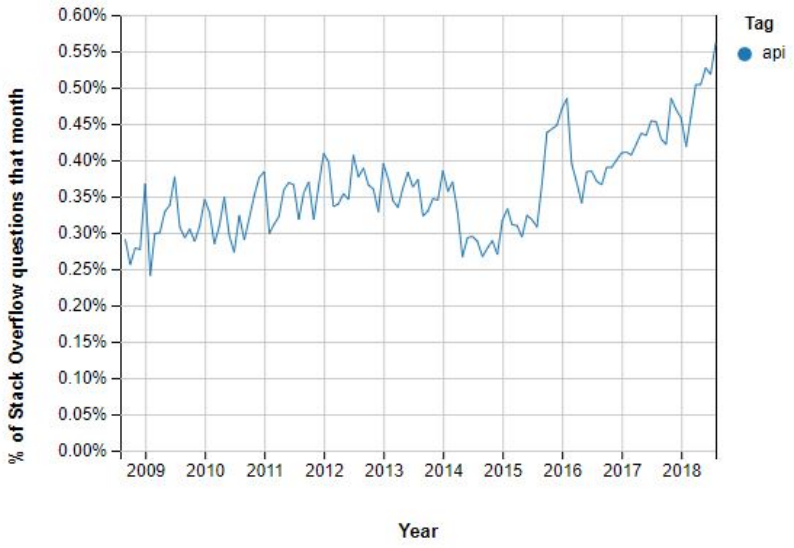

Abbildung 1.2: Trendentwicklung des Suchworts „API“ auf Stackoverflow.com

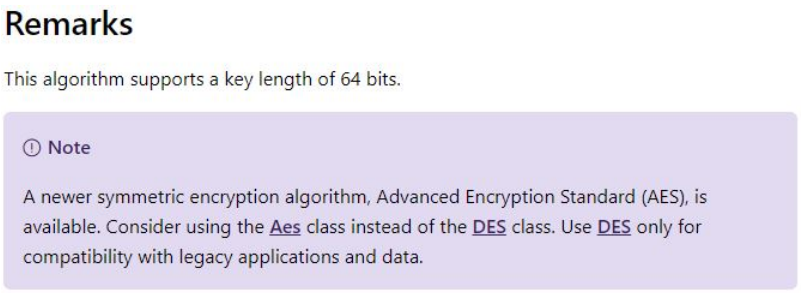

Abbildung 1.3: Hinweis zur veralteten DES-Verschlüsselung in der API Dokumentation von Microsoft

> Es wird festgehalten: Programmierer sind nicht in der Lage Krypto-APIs ordnungsgemäß zu benutzen. API Hersteller zeigen Schwächen, ihre Dokumentationen verständlich bzw. fehlerfrei zu halten.

Wie bereits erwähnt, treten fehlerhafte Verwendungen von Krypto-APIs in Anwendungen recht häufig auf. Eine Quote von bis zu 95% ist alarmierend. Es stellen sich direkt die Fragen, wieso die Entwickler derartige Schwierigkeiten besitzen und wie ihnen zu helfen ist, diese Fehler zu vermeiden. Die erste dieser Fragen wurde bereits durch viele Forschungen beantwortet (s. Abschnitt 3.2). Die andere Frage hingegen bleibt dabei teils unbeantwortet oder wird nur theoretisch diskutiert. Praktische Ansätze, die Entwickler bei ihrer Arbeit unterstützen, sind kaum vorhanden. Diese Masterarbeit beschäftigt sich daher mit der konkreten Umsetzung eines Hilfswerkzeug für Programmierer, welches die fehlerhafte Verwendung einer Krypto-API aufspüren kann. Es gibt viele verschiedene Arten von Hilfswerkzeugen, darunter auch ein Static Code Analysis Tool (SCAT), welches für diese Arbeit ausgewählt wurde. Ein SCAT erlaubt eine Codeanalyse im frühen Entwicklungsstadium, da der zu untersuchende Code nicht lauffähig sein muss. Die Codeanalyse kann daher schon durchgeführt werden, während der Programmierer Quelltext schreibt. Das Einsetzen und korrekte Bedienen von Hilfswerkzeugen, ist in der heutigen Zeit unablässig. Eines der ersten SCATs, soweit dem Autor bekannt, welches sich ausschließlich auf die korrekte Ver-

wendung von Krypto-APIs spezialisiert, wurde erst Ende des Jahres 2017 vorgestellt und Mitte 2018 veröffentlicht. Automatisierungsmöglichkeiten, wie sie auch statische Codeanalyseprogramme bieten, sind allein schon wegen der ständig steigernden Komplexität und Umfang der Programmcodes[17] empfehlenswert. Personen-basierte Inspektionsprozesse unterliegen der Schwäche des Menschen, Dinge zu übersehen oder falsch zu bewerten.

> Es wird festgehalten: Ausgereifte und praktische Lösungsansätze zur Unterstützung der Nutzung von Krypto-APIs sind aktuell kaum verfügbar.

Viele Forschungen konzentrieren sich überwiegend oder ausschließlich auf die Betrachtung der Programmiersprache Java. Die neusten Erweiterungen rund um C#, argumentieren jedoch für eine genauere Betrachtung dieser Sprache. War C# früher noch für das Betriebssystem Windows mehr oder weniger exklusiv, so ist sie heutzutage auf allen gängigen Systemen, darunter auch Android, iOS, Windows, macOS oder Linux vertreten. Entwicklungsmethodiken wie Cross-Platfrom Development sind bei C# möglich. Hier ist das Risiko hoch, dass eine einzige falsche Verwendung einer Krypto-API in einer plattformunabhängigen Softwarekomponente, gleich eine Bedrohung für alle Plattformen darstellt, auf denen die Anwendung ausgeführt wird. Die Betrachtung dieser Programmiersprache, wie es diese Arbeit sich zum Ziel gesetzt hat, ist daher von hoher Relevanz.

> Es wird festgehalten: Sämtliche Programmiersprachen, mit Ausnahme von Java werden in Forschungen nicht berücksichtigt, obwohl diese an Relevanz gewinnen.

Aus der unzureichenden Abdeckung einer in der Praxis relevanten Programmiersprache (C#), der geringen Verfügbarkeit von Werkzeugen für Programmierer, Fehler bei der Verwendung von Krypto-APIs zu erkennen und zu beheben und der Beobachtung, dass viele – wenn nicht sogar die meisten – Anwendungen und Tutorials fehlerhaft sind, entsteht die Motivation mittels eines eigens entwickelten statischen Codeanalyseprogramms, die hier beschriebenen Umstände auszubessern.

1.2 Beitrag zur Wissenschaft

Es gibt viele Methoden eine korrekte Verwendung von Krypto-APIs sicherzustellen oder diese zu erleichtern. Statische Codeanalyseprogramme bieten sich besonders an, da diese von den Entwicklern bereits während des Programmierens eingesetzt werden können. Sie bieten eine Chance, den Entwickler bei der Verwendung dieser APIs zu unterstützen und frühzeitig Fehler aufzudecken. Aus der für diese Arbeit untersuchten Literatur, konnte bisher nur ein solch spezialisiertes Tool für die Programmiersprache Java identifiziert werden – CogniCrypt. Andere relevante Programmiersprachen, mit Ausnahme noch von C/C++, darunter auch C# sind weitestgehend in sämtlichen Forschungsthemen dieser Art unterrepräsentiert. Mit der Entwicklung eines eigenen SCAT mit dem Namen *Sharper Crypto-API Analysis* für die Programmiersprache C# und den damit verbundenen Technologien, soll ein Ausgleich in der Fachliteratur

[17] https://informationisbeautiful.net/visualizations/million-lines-of-code/, Zugriff: 11.12.2018

geschaffen werden. Ebenfalls ermöglicht die Verwendung von C# eine differenzierte Auskunftsgabe über vorhandene Parallelen und Unterschiede der verwendeten Programmiersprachen, APIs und der Entwicklung eines SCAT.

Das marktfähige Entwickeln eines SCAT ist im Rahmen einer Abschlussarbeit nicht realistisch. Sharper Crypto-API Analysis setzt sich daher zum Ziel, wesentliche Codeanalysen und eine Architektur vorzustellen und zu implementieren, sodass bekannte Anwendungsfehler mit einer Krypto-API bereits identifiziert werden können, aber auch eine Weiterentwicklung des SCAT möglich ist.

1.3 Ziele

Zentrales Ziel dieser Masterarbeit ist es ein integriertes, statisches Codeanalyseprogramm namens Sharper Crypto-API Analysis für C# und Visual Studio, eine Integrierte Entwicklungsumgebung (IDE) für Windows, zu entwickeln. Dieses Tool soll den Quellcode einer Anwendung auf Fehler oder Verstöße bei der Verwendung der .NET Krypto-API aus dem Namespace `System.Security.Cryptography` untersuchen. Werden unsichere Codestellen gefunden, soll das Tool entsprechende Informationen bereitstellen, die erläutern, warum der Code fehlerhaft ist und zudem einen Lösungsvorschlag bereitstellen. Die folgende Auflistung beschreibt die einzelnen Teilziele, die bei der Entwicklung von Sharper Crypto-API Analysis erfüllt werden sollen:

Ziele hinsichtlich der Funktionalität:

- Erkennen fehlerhafter/unsicherer Verwendungen der .NET Krypto-API
- Warnen des Benutzers bei unsicherem Code
- Unsichere Codestellen anzeigen
- Bereitstellen von Informationen über Strukturen und Methoden der .NET Standard Krypto-API
- Bereitstellen von zusätzlichen Informationen über den analysierten Code
- Vorschlagen von Codeverbesserungen
- Automatische Codegenerierung der Verbesserungen
- Erweiterbarkeit des Regelwerks durch externe Anbieter
- Erweiterbarkeit des Regelwerks durch den Benutzer
- Codegenerierung zur einfacheren Benutzung der .NET Krypto-API

Ziele hinsichtlich der Gebrauchstauglichkeit

- Anpassung der Fehlerraten und Warnstufen
- Integration in die IDE Visual Studio
- Verwendung von UI-Elementen der IDE, um eine Vertrautheit mit dem Nutzer zu schaffen (einfachere Lernkurve)

- Keine/wenige Voraussetzungen mit Kryptographie, um das Tool zu verwenden
- Möglichst hohe Soundness, um Schwachstellen in der Software zu vermeiden
- Möglichst hohe Vollständigkeit, um irritierende False Positives zu vermeiden
- Warnungen und Berichte müssen auf dem derzeit vorhandenen Anwendungscode basieren
- Wartbarkeit durch Aktualisierungsmöglichkeiten

Ziele hinsichtlich der IT-Sicherheit:

- Berücksichtigung neuster Erkenntnisse zu einzelnen kryptographischen Verfahren
- Codegenerierungen müssen aktuellen Sicherheitsstandards entsprechen

Anhand eines Referenzprojekts, das zusätzlich zu dem SCAT entwickelt wird, soll die Funktionalität des Tools evaluiert und demonstriert werden. Dieses Referenzprojekt soll ein Softwareprogramm sein, um Daten auf dem Rechner zu verschlüsseln, welches die Krypto-APIs jedoch absichtlich falsch benutzt. Die eingebauten Fehler werden entsprechend im Code dokumentiert. Das SCAT soll letzten Endes in der Lage sein, diese Fehler zu identifizieren.

1.4 Aufbau

Diese Masterarbeit ist im Folgenden in fünf Abschnitte eingeteilt. Das nächste Kapitel beschreibt grundlegende, themenbezogene Begriffe, die zum besseren Verständnis der Arbeit führen. In Kapitel 3 werden aktuelle Forschungsergebnisse der Fachliteratur im Kontext dieser Arbeit vorgestellt. In Kapitel 4 erfolgt eine genaue Dokumentation des Entwicklungsprozesses von Sharper Crypto-API Analysis, inklusive einer Beschreibung der Vorgehensweise, der Spezifikation von erforderlichen Anforderungen und Codeanalysen sowie der Implementation. Kapitel 5 diskutiert die gewonnenen Erkenntnisse und führt eine Bewertung der erreichten Ziele durch. Zum Schluss wird in Kapitel 6 ein Ausblick auf künftige Funktionen und Untersuchungsmöglichkeiten gegeben und ein Fazit gezogen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für diese Arbeit grundlegenden Begriffe ausführlich erklärt und sofern erforderlich, für den Kontext dieser Arbeit definiert. Durch kontextbezogene Beispiele wird dabei immer der Bezug zur Verwendung von Krypto-APIs oder der in dieser Arbeit verwendeten Programmiersprache C# genommen.

Zunächst wird im Abschnitt 2.1 der zentrale Begriff Krypto-API für die weitere Verwendung in dieser Arbeit genau beschrieben und definiert. Eine ausführliche Begriffsbeschreibung eins statischen Codeanalyseprogramms, wie es in dieser Arbeit entwickelt wird sowie eine kurze Auflistung aktuell erhältlicher SCATs liefern die Abschnitte 2.2 und 2.3. Eine Begründung der genaueren Betrachtung eines SCAT in dieser Arbeit, erfolgt nach dem Vergleich dieses mit weiteren Testverfahren der Softwareentwicklung in Abschnitt 2.4. Abschließend wird in Abschnitt 2.5 der Einfluss des Faktor Mensch in der Softwareentwicklung herausgestellt.

2.1 Krypto-APIs

Das Wort Krypto-API kombiniert die Begriffe Kryptographie und API. Die Kryptographie ist ein Teilgebiet der Kryptologie, die Lehre und Wissenschaft der Geheimschrift. Sie beschäftigt sich mit der Chiffrierung von Informationen (Paar & Pelzl, 2016, S. 2). Im Bezug auf Softwareanwendungen können mittels kryptographischer Verfahren unter anderem Dateien und Daten verschlüsselt bzw. signiert werden. Zur Verwendung dieser Verfahren werden sie in Softwarekomponenten implementiert und können so von verschiedenen Anwendungen verwendet werden.

Eine API kann als Schnittstellenspezifikation verstanden werden, die eine Menge an Funktionen anhand von Eingabeparametern und Ausgabewerten definiert (Bloch, 2014, S. 34). Es handelt sich also viel mehr um eine Dokumentation als um eine Implementation. Die Implementation einer API wird bei prozeduralen oder objektorientierten Programmiersprachen häufig als Bibliothek bezeichnet. Im konkreten Beispiel der Programmiersprache C# existiert auch der Ausdruck *Assembly*. Im Alltag werden die Begriffe API und Bibliothek jedoch synonym verwendet[1].

Um diesem Umstand zu entsprechen, wird in dieser Arbeit eine Krypto-API als eine Sammlung von Implementationen kryptographischer Verfahren verstanden, die mittels einer dokumentierten Schnittstelle aufgerufen werden können.

[1] https://simplicable.com/new/library-vs-api, Zugriff: 18.10.2018

2.2 Statische Codeanalyseprogramme

2.2.1 Allgemeines und Herkunft

Die statische Codeanalyse selbst ist als eine spezielle Form bzw. Teilmenge der Codeinspektionen, dem Prozess, bei dem neuer oder geänderter Code von einer Instanz gegengelesen wird, zu verstehen und beschreibt das Analysieren einer Software, ohne dass diese ausgeführt werden muss. Die Analyse kann sich sowohl auf den Quellcode der Anwendung als auch, falls vorhanden, auf den Zwischencode der Anwendung beziehen (Novak, Krajnc & Žontar, 2010, S. 418). Untersuchungen belegen, dass dieser Prozess die Qualität der Software positiv beeinflussen kann (Baker, 1997; McIntosh, Kamei, Adams & Hassan, 2014). Sie unterscheidet sich von der dynamischen Codeanalyse insofern, als das der Code nicht ausgeführt werden oder gar lauffähig sein muss. Der Einsatz der statischen Codeanalyse eignet sich daher schon früh, noch vor der Benutzung von Modultests (Chelf & Ebert, 2009, S. 96), in der Entwicklungsphase der Softwareimplementation. Das SCAT ist eine Software, die diese Form der Analyse durchführt und den Entwickler bei der Codeinspektion unterstützt. Die ersten Tools basierten grundsätzlich auf dem `grep` Befehl für UNIX (Baca, Petersen, Carlsson & Lundberg, 2009, S. 805; Chess & McGraw, 2004, S. 77). Im weiteren Verlauf wurden diese Tools jedoch um Technologien erweitert, die auch ein Kompiler verwendet. Untersuchungen konnten zeigen, dass dadurch die Fehler- und Erkennungsraten dieser Programme im Vergleich zu grep-basierten Programmen verbessert werden konnten (J. Viega, Bloch, Kohno & McGraw, 2000). Einen vergleichsweise hohen Bekanntheitsgrad erreichte das C-Analyseprogramm *Lint* von Stephen Johnson. Johnson begründete das Entwickeln von Lint mit der Arbeitsweise der damaligen Kompiler[2]. Deren Hauptaufgabe sei es C-Code schnell und effizient in ausführbare Programme zu übersetzen. Diverse Operationen wie das Typumwandeln (engl. casting) seien bei der Übersetzung nicht überprüft worden (S. C. Johnson, 1978). Moderne Analyseprogramme funktionieren, vereinfacht ausgedrückt, ähnlich wie Antivirusprogramme, indem sie nach festgelegten Mustern im Code suchen oder versuchen den Datenfluss des Programms oder seinen Teilen zu analysieren (Novak et al., 2010).

2.2.2 Erkennungsklassen

Neben den gerade beschriebenen verschiedenen Funktionsweisen dieser Analysetools, unterscheiden sie sich auch in ihren Anwendungsszenarien. Generell kann man zwischen Allzweckanalyseprogrammen und spezialisierten Analyseprogrammen unterscheiden. Allzweckanalyseprogramme versuchen gängige und immer wieder auftretende Programmierfehler aufzudecken und sind daher meist von der Anwendungsdomäne der Software unabhängig. Spezialisierte Analyseprogramme versuchen Programmierfehler aufzudecken, die häufig in einer konkreten Anwendungsdomäne auftreten. Mittels sogenannter Erkennungsklassen kann man beispielsweise das Anwendungsgebiet eines SCAT grob einordnen. Einige dieser Erkennungsklassen werden in diversen Artikeln beschrieben (Chelf & Ebert, 2009; Novak et al., 2010; WASC, 2013):

1. Teilen durch Null

[2]Der Funktionsumfang der Kompiler ist heutzutage größer geworden, sodass einige von ihnen zum Teil in der Lage sind, die von Johson erwähnten Mängel zu beheben.

2. Speicherlecks
3. Nullzeiger Dereferenzierung
4. Pufferüber- und unterläufe
5. Falsche Typumwandlung
6. Unbenutzter oder nicht erreichbarer Code

Weitere Erkennungsklassen sind nicht ausgeschlossen. Wie schwerwiegend der Fehler aus einer Erkennungsklasse ist, kann pauschal nicht beantwortet werden, sondern steht häufig im Kontext der verwendeten Programmiersprache. Listing 2.1 zeigt den Ausschnitt eins C# Programms, welches einen Fehler aus der Erkennungsklasse *Nummer 5* beinhaltet. Die C#-Laufzeitumgebung wird in der zweiten Zeile eine `InvalidCastException` werfen und das Programm, sofern nicht anders behandelt, terminieren.

```
1 object t = "123";
2 var i = (int) t;
3 Console.WriteLine((int) i);
```

Listing 2.1: Falsche Typumwandlung in C#

Ein ähnlicher Code in C++, zu sehen in Listing 2.2, wirft keine Ausnahme und terminiert auch das Programm nicht. Stattdessen wird in der zweiten Zeile die Speicheradresse der Variable ausgegeben. Das C++ Programm gibt somit keinen eindeutig bestimmbaren Wert aus und kann sich möglicherweise nicht deterministisch verhalten.

```
1 auto i = (int) "123";
2 printf("%d", (int) i);
```

Listing 2.2: Falsche Typumwandlung in C++

Ob nun ein gesicherter Programmabsturz wie im C# Beispiel oder eine mögliche inkonsistente Programmfortsetzung wie im C++ Beispiel ein schwerwiegenderer Fehler ist und zu einer Verwundbarkeit führen kann, hängt von den Anforderungen an das Programm ab. Ein SCAT kann daher nicht mit Gewissheit evaluieren, ob eine Schwachstelle vorliegt, sondern lediglich einen Hinweis darauf geben, dass der untersuchte Code Schwächen besitzen könnte.

2.2.3 Erkennungsklassen der Nutzung von Krypto-APIs

Aus den bisher aufgelisteten Erkennungskassen lassen sich jedoch nur schwer Nutzungsmöglichkeiten erkennen, die die Analyse von Code der Nutzung von Krypto-APIs beinhalten. Zwar wird in den Evaluationskriterien eines SCAT des WASC die Erkennungsklasse „Unsichere Kryptographie“ (WASC, 2013, S. 8) aufgeführt, jedoch nicht detaillierter beschrieben. Der Grund für diesen Umstand ist leicht identifiziert: All diese Erkennungsklassen sind darauf ausgelegt, möglichst viele Softwareprojekte zu unterstützen. Der Gebrauch von Krypto-APIs, oder allgemeiner formuliert, einer beliebigen API, ist je nach Software eine sehr spezifische Aufgabe, die stark vom

Kontext der Anwendung abhängt. Wie bereits festgestellt, haben Allzweckanalyseprogramme jedoch Schwierigkeiten diesen Kontext nachzuvollziehen, um eine ausreichende Aussage über die Nutzung von APIs zu treffen[3]. Bei SCATs, die sich auf die Analyse zur Nutzung einer API spezialisieren, werden entsprechende Funktionalitäten implementiert, die versuchen den Anwendungskontext anhand des Codes oder anderen Parametern nachzuvollziehen. Im Falle eines SCAT zur Analyse der Nutzung von Krypto-APIs ist es beispielsweise wichtig, dass das Analyseprogramm die zu untersuchende Krypto-API kennt. Damit ist gemeint, dass das Tool die einzelnen Funktionen, ihre Rückgabewerte und Eingabeparameter sowie das Verhalten der Funktionen versteht und bei der Codeanalyse berücksichtigt. Nachfolgend werden einige Erkennungsklassen vorgeschlagen, die sich speziell auf diese Art der Analyse richten und die eingangs erwähnte Erkennungsklasse „unsichere Kryptographie“ konkretisieren:

- Schwache Verschlüsselungsalgorithmen
- Schwache Hashingalgorithmen
- Schwache Konfiguration der benutzten API
- Verstoß gegen die API-Spezifikation
- Auslesbare Informationen

Im Abschnitt 4.4 geht auf die hier vorgeschlagenen Erkennungsklassen im Detail ein.

2.3 Verfügbare statische Codeanalyseprogramme

Vor der Entwicklung eines neuen SCAT ist es sinnvoll sich über bereits existierende Produkte zu informieren. Folgende nicht vollständige Auflistung gibt eine kompakte Auskunft über das Funktionsangebot und die Stärken und Schwächen anderer SCATs. Anhand dieser Informationen kann entschieden werden, welche Funktionalitäten, Fokus und Alleinstellungsmerkmale das hier entwickelte neue SCAT besitzen soll.

1. **Sonarlint**[4] ist eine quelloffene Sammlung von SCAT-Frontends für IDE Erweiterungen. Derweilen werden die IDEs IntelliJ, Eclipse, Visual Studio (VS), VS Code (VSC) und Atom unterstützt. Je nach IDE und Programmiersprache werden verschiedene Backends zur Codeanalyse verwendet. **SonarC#**[5] ist beispielsweise das Backend für die Visual Studio Erweiterung und basiert auf der Compilerplattform *Roslyn*. Je nach gewählter Programmiersprache wird der Quellcode auf bis zu 300 verschiedene Regeln geprüft. Viele davon können den Allzweckanalysen zugeordnet werden, einige jedoch beziehen sich auch speziell auf die Verwendung von Krypto-APIs.

[3] Es sei angemerkt, dass es natürlich dem Hersteller des SCAT obliegt, ob und wie viele Entwicklungsressourcen in die einzelnen Erkennungsklassen fließen. Die Übergänge zwischen einem allzweck und einem spezialisierten SCAT können daher fließend sein.

[4] https://www.sonarlint.org, Zugriff: 30.12.2018

[5] https://github.com/SonarSource/sonar-csharp, Zugriff: 30.12.2018

2. **Checkstyle**[6] ist ein regelbasiertes, quelloffenes Allzweckanalysewerkzeug für Java. Mit ihm lassen sich unter anderem Codekonventionen, Klassenaufbau und Codestyle untersuchen. Checkstyle ist modular erweiterbar, erfordert jedoch einen gewissen Konfigurationsaufwand.

3. **FindBugs**[7] ist ein quelloffenes SCAT für Java, welches von der University of Maryland entwickelt wurde. Sein Nachfolger ist **SpotBugs**[8]. Im Gegensatz zu Sonarlint und Checkstyle untersucht FindBugs nicht den Quellcode, sondern den Bytecode eines Programms. Das Programm selbst muss jedoch für die Analyse nicht ausgeführt werden. FindBugs kann über 300 verschiedene Fehler und Auffälligkeiten finden, die sowohl Allzweckanalysen als auch spezielleren Analysen zugeordnet werden können. Als Beispiel für spezielle Analysen sind unter anderem das Identifizieren möglicher `null` Rückgabewerte bei der `Clone()` Methode zu nennen. FindBugs ist sowohl als eigenständiges Tool, wie auch als Plugin für diverse IDEs wie Eclipse verfügbar.

4. **Gendarme**[9] ist, ähnlich wie FindBugs, ein quelloffenes SCAT, welches den Bytecode von .NET bzw. Mono Anwendungen untersucht. Gendarme ist ein regelbasiertes Tool, das auf dem Monoprojekt namens *Cecil* basiert, welches den Bytecode der Common Language Infrastructure (CLI) Spezifikation (ECMA International, 2012) untersuchen kann.

5. **Resharper**[10] ist eine Erweiterung für die IDE Visual Studio. Resharper konzentriert sich primär auf das Einhalten von Codekonventionen und das Finden und Korrigieren von Fehlern oder Ungenauigkeiten im Quellcode. Die Erkennungsklassen dieses Programms sind daher eher im Bereich der Allzweckanalyse einzuordnen, auch wenn sich benutzerdefinierte Regeln hinzufügen lassen. Resharper ist ein kommerzielles Produkt. Neben den Programmiersprachen C#, Visual Basic und C++ werden auch JavaScript oder HTML unterstützt. Neben Codeanalysen bietet Resharper auch sogenannte Quick-Fixes an.

Die Tabelle 2.1 führt einen kurzen Vergleich, der hier vorgestellten SCATs vor. Das SCAT CogniCrypt wird detaillierter im Abschnitt 3.5 beschreiben und ist daher nicht in dieser Auflistung aufgeführt.

2.4 Statische Codeanalyse im Vergleich mit anderen Maßnahmen zur Qualitätssicherung

Das Testen von Software spielt in der Anwendungsentwicklung eine bedeutende Rolle. Etwa 50% der Entwicklungszeit und Entwicklungskosten werden für das Testen aufgebracht (Myers, Sandler & Badgett, 2011, S. ix). Für sicherheitskritische Anwendung kann dieser Prozentsatz deutlich größer sein (Ammann & Offutt, 2008, S.

[6] http://checkstyle.sourceforge.net, Zugriff: 30.12.2018
[7] http://findbugs.sourceforge.net, Zugriff: 30.12.2018
[8] https://spotbugs.github.io, Zugriff: 30.12.2018
[9] http://www.mono-project.com/docs/tools+libraries/tools/gendarme/, Zugriff: 30.12.2018
[10] https://www.jetbrains.com/resharper/, Zugriff: 30.12.2018

Tabelle 2.1: Vergleich einiger verfügbarer SCATs

	Sonarlint	Checkstyle	FindBugs	Gendarme	Resharper
Plattform	u.a. Java und C#	Java	Java Bytecode	CIL	u.a. C# und C++
Einsatzgebiete	Hauptsächlich allgemeine Programmierfehler	Codekonventionen	Allgemeine Programmierfehler	Allgemeine Programmierfehler	Refactoring
Quick-Fixes	vorhanden	vorhanden	keine	keine	vorhanden
Codegenerierung	keine	keine	keine	keine	keine
IDE-Integration	u. a. VS, VSC	u.a. IntelliJ, Eclipse	u.a. IntelliJ, Eclipse	keine	VS
Kosten	kostenfrei	kostenfrei	kostenfrei	kostenfrei	kostenpflichtig

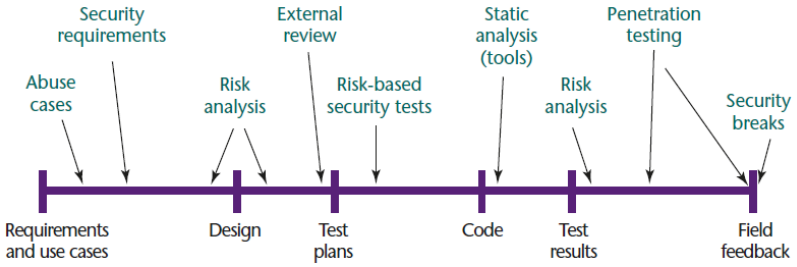

Abbildung 2.1: Secure Software Development Life Cycle nach McGraw, 2004

10). Mit dem Testen der Software soll sichergestellt werden, dass die Software genau das erledigt, was die Entwickler vorgesehen haben (Myers et al., 2011, S. 2). Um potentielle Fehler so früh wie möglich zu erkennen, empfiehlt sich der Einsatz geeigneter Testmethoden schon während der Entwicklung. Abbildung 2.1 zeigt den Secure Software Development Life Cycle (SSDLC) nach McGraw, 2004. Die violette Zeitachse beschreibt das aus der Softwareentwicklung bekannte Wasserfall-Modell (Benington, 1983, S.3; Kleuker, 2018, S.27). Die mit Pfeilen gekennzeichneten Begriffe oberhalb der Zeitleiste sind bewährte Verfahren, die zu dem angezeigten Zeitpunkt in der Entwicklung angewendet werden. Erkennbar ist, dass neben der statischen Codeanalyse noch weitere Verfahren existieren, um eine Qualitäts- und Sicherheitsprüfung der Software in der Entwicklungsphase durchzuführen. In diesem Abschnitt werden zwei dieser anderen Verfahren, die Modultests und der Penetrationstest, kurz vorgestellt. Anschließend werden sie und die toolbasierte statische Codeanalyse anhand wichtiger Kriterien miteinander verglichen, um so die Stärken, aber auch Grenzen oder Schwächen der einzelnen Verfahren zu beleuchten. Gleichermaßen bietet dieser Abschnitt auch die Begründung, warum sich in dieser Arbeit für die statische Codeanalyse als Qualitätssicherungsverfahren entschieden wurde.

Penetrationstests

Laut Arkin, Stender und McGraw, 2005, S. 84 ist der Penetrationstest die am häufigsten angewendete Methofde der Qualitätssicherung. Engebretson, 2013, S. 1 definiert dieses Verfahren als erlaubten Versuch, Schwachstellen in Computersystemen aufzudecken und diese auszunutzen. Die Erkenntnisse des Tests tragen dazu bei, dass der Software zusätzliche Sicherheitsmechanismen hinzugefügt werden. Grundsätzlich unterscheidet man zwischen White- und Blackbox-Penetrationstest. Bei Letzterem sind dem Tester der Quellcode der Anwendung oder weitere Zusatzinformationen nicht bekannt. Eine Garantie, dass möglicherweise offensichtliche Schwachstellen effizient gefunden werden, gibt es daher nicht.

Modultests

Während Penetrationtests eher am Ende des Softwareentwicklungszyklus durchgeführt werden, finden Modultests (engl. unit tests) auch während der Programmierphase stetig Anwendung. Beim Unittesting wird die Funktionalität der zu entwickelnden Software automatisch anhand von Annahmen an das Ergebnis eines Moduls geprüft (Osherove, 2015, S. 11). Liefert das Modul nicht das angenommene Ergebnis, kann ein Fehler der Software angenommen werden. Unter einem Modul können verschiedene Komponenten einer Software verstandenen werden. Eine präzise Einordnung des Begriffs ist nicht möglich. Betrachtet man lediglich die benutze Programmiersprache, könnte man unter einem Modul in objektorientierten Sprachen eine Klasse und in prozeduralen Sprachen eine Funktion verstehen[11]. Andere Einordnungen sind jedoch auch möglich (Osherove, 2015, S. 4f).

2.4.1 Vergleichskriterien

Anhand der aufgestellten Beschreibungen dieser Testverfahren lassen sich Kriterien ableiten, mit denen ein Vergleich durchgeführt werden kann:

- **Durchführungszeitpunkt:** Dieses Kriterium beschreibt, zu welchem Zeitpunkt ein Testverfahren üblicherweise im Softwareentwicklungsprozess durchgeführt wird.
- **Durchführungsressourcen:** Für die einzelnen Testverfahren sind unterschiedliche Arbeitsschritte notwendig. Welche Ressourcen hierfür aufgewendet werden, beschreibt dieser Punkt.
- **Benötigte Durchführungskenntnisse:** Für eine effiziente Durchführung sollten Softwaretestverfahren nach genau beschriebenen Prozessen ausgeführt werden. Dieses Kriterium beschreibt, welche Kenntnisse von der durchführenden Person benötigt werden.
- **Benötigte Fachkenntnisse im Bereich IT-Sicherheit:** Bezogen auf das Thema dieser Arbeit, ist die Frage nach den benötigten Kenntnissen im Bereich der IT-Sicherheit von wichtiger Bedeutung. Nicht alle Entwickler und

[11] https://martinfowler.com/bliki/UnitTest.html, Zugriff: 01.01.2019

Qualitätsprüfer sind ausgewiesene Experten in diesem Thema. In welchem Maße Kenntnisse zur Durchführung der Testmethode in diesem Bereich notwendig sind, beschreibt dieser Punkt.

- **Fehlerbehebungsressourcen:** Ist ein Fehler identifiziert, muss dieser auch behoben werden. Je nach aktuellem Standpunkt im Software Development Life Cycle (SDLC) variieren die damit verbundenen Ressourcen.
- **Fehlererkennungsrate:** Ein Testverfahren, das wenig Resultate liefert, besitzt unter Umständen ein hohe False Negative Rate. Dieses Kriterium gibt Auskunft über die zu erwartende Quantität der ermittelten Fehler.
- **Aussagekraft:** Dieses Kriterium beschreibt die Qualität der gefundenen Softwarefehler.

2.4.2 Vergleich

Ein zentraler Vergleichsaspekt ist der Durchführungszeitpunkt der verschiedenen Testverfahren. Weitere Kriterien wie die Aussagekraft der Resultate oder auch die Fehlererkennungsressourcen sind von diesem Zeitpunkt abhängig. Je später ein Fehler bekannt wird, desto höher steigen die Kosten und der Aufwand, um den Fehler zu beheben (C. Jones, 2008). Sowohl Modultests, als auch die statische Codeanalyse werden bereits in der Implementierungsphase der Software durchgeführt. Penetrationstest können hingegen effektiv erst in den späten Testphasen angewendet werden. Auch die statische Codeanalyse kann in dieser Testphase erneut zum Einsatz kommen. Die benötigten Fehlerbehebungsressourcen sind daher bei Modultests und der statischen Codeanalyse geringer, als bei den Penetrationstests. Natürlich spielen hier jedoch noch weitere Faktoren wie Codequalität oder Codekenntnis eine Rolle und können daher auch bei der statischen Codeanalyse oder den Modultests den Ressourcenaufwand erhöhen (Kononenko, Baysal & Godfrey, 2016).

Auch bei den benötigten Durchführungsressourcen und Fachwissen unterscheiden sich die Testverfahren. Eine statische Codeanalyse, die mit einem Programm durchgeführt wird, hat den Vorteil, eines geringen Zeit- und Arbeitsaufwand bei der Fehlersuche. Die notwendigen Ressourcen der Bewertung der Fehlermeldungen sind hingegen abhängig vom Kenntnisstand desjenigen, der diese Bewertung durchführt und der Anzahl der gefundenen Codeanalysen. Ein Experiment von Uwano, Nakamura, Monden und Matsumoto, 2006 zeigte jedoch, dass sich die Entwickler nicht genügend Zeit nehmen diese Analyse durchzuführen. Sowohl Modul-, als auch Penetrationstests verursachen einen wesentlich größeren Aufwand an Zeit und Arbeit. Bei Modultests wird unter anderem ein Mehrzeitaufwand von 16% erwartet (George & Williams, 2004, S. 340). Auch der Arbeitsaufwand kann enorm sein, da für sämtliche Funktionalitäten und Parameter Tests programmiert werden müssen, eine Arbeit, die sich Entwickler gerne sparen würden (Daka & Fraser, 2014, S. 210). Bei den Blackbox Penetrationstest fließt ein erheblicher Teil an Zeit und Arbeit allein in die Vorbereitung des Tests und Einarbeitung in das anzugreifende System. Ein sehr hohes Expertenwissen, insbesondere auch im Bereich der IT-Sicherheit, wird hier beim Tester vorausgesetzt. Dazu zählt nicht nur das Wissen, was mögliche Schwachstellen sein könnten, sondern auch wie diese anzugreifen sind. Aus diesem Grund werden Penetrationstest auch am häufigsten falsch durchgeführt (Arkin et al., 2005, S. 87).

Die Quantität und Qualität der identifizierten Fehler ist von entscheidender Bedeutung und rechtfertigt erst den Einsatz eines Testverfahrens. Zunächst sei bemerkt, dass das Ergebnis dieser Kriterien stark abhängig von der Durchführung des Testverfahrens ist. Dennoch können Aussagen darüber getroffen werden, welche Ergebnisse zu erwarten sind. Im Vergleich zu den anderen beiden Verfahren, können statische Codeanalysen die höchsten Fehlererkennungsraten aufweisen. Ein entscheidender Nachteil, wie sich in Abschnitt 3.3.2 herausstellen wird, ist jedoch die hohe Anfälligkeit von False Positive Analysen. Das Resultat einer statischen Codeanalyse muss daher genau evaluiert werden, um False Positves von True Prositives Fällen zu unterscheiden. Die Fehlererkennungsrate bei Modultests ist hingegen direkt abhängig von der Anzahl der Tests, die für ein Modul geschrieben wurden. Wurde ein Modul oder einzelne Modulteile, wie bestimmte Eingabeparameter, nicht durch einen Test abgedeckt, so kann auch hier kein Fehler ermittelt werden. Untersuchungen zeigen, dass Entwickler Schwierigkeiten besitzen zu entscheiden, welchen Code sie mit Tests abdecken müssen (Daka & Fraser, 2014, S. 209). Modultests sind daher anfällig für False Negatives. Ähnlich verhält es sich mit den Penetrationstests. Die Fehlererkennungsrate ist hierbei besonders beim Blackbox Verfahren sehr stark von den Fähigkeiten des Testers abhängig. Da Penetrationstests häufig bereits am fertiggestellten System angewendet werden, haben gefundene Schwachstellen jedoch eine hohe Aussagekraft und stellen eine reelle Bedrohung für das System dar.

Eine bessere Veranschaulichung des hier aufgestellten Vergleich ist in Tabelle 2.2 dargestellt.

2.4.3 Bewertung

Als wichtige Beobachtung aus dem hier gezogenen Vergleich geht hervor, dass sich die einzelnen Testverfahren gegenseitig ergänzen. Mit zunehmendem Entwicklungsfortschritt werden zwar die Anforderungen an das Testverfahren größer (Novikov, Ivutin, Troshina & Vasiliev, 2017), jedoch sind die daraus erzielten Ergebnisse präziser und detaillierter und können somit die Sicherheit der Software besser, wenn auch niemals vollständig, bewerten. Die in der Einleitung zitierte Aussage von Dijkstra besagt, dass das Verhalten einer Software nicht verifiziert, sondern lediglich falsifiziert werden kann. Vereinfacht ausgedrückt bedeutet diese Aussage, dass selbst bei Anwendung aller möglichen Testverfahren, eine 100 prozentige Fehlerfreiheit nie sichergestellt werden kann. Weitere Untersuchungen belegen diese Aussage (Basili & Selby, 1987; Bertolino, 2007).

Sucht man nun ein Testverfahren, welches Fehler beim Umgang mit Krypto-APIs untersucht, kann dieser Vergleich eine Entscheidungsgrundlage bilden. Es stellt sich heraus, dass toolbasierte statische Codeanalysen einen geeigneten Kandidaten abgeben. Das erforderliche Expertenwissen im Bereich IT-Sicherheit eines Programmierers muss nicht besonders ausgeprägt sein. Wie in Abschnitt 3.2 dargestellt wird, verwenden Entwickler häufig Code, der ihnen vorgeschlagen wird und integrieren ihn in ihre Anwendungen. Eine Bewertung, ob dieser Codeausschnitt sicher ist, erfolgt meistens nicht. Ein statisches Codeanalyseprogramm mit der Spezialisierung auf der Verwendung von Krypto-APIs kann direkt bei der Codeeingabe überprüfen, ob der Programmierer schwachen Code benutzt. Testfälle, wie sie bei den Modultests erforderlich sind, die weiteres Fachwissen erfordern, muss er in diesem Fall nicht schreiben. Der geringe Arbeitsaufwand diese Codeanalyse durchzuführen (im besten Fall

Tabelle 2.2: Vergleich einzelner Softwaretestverfahren

Kriterien	Toolunterstützte statische Codeanalyse	Modultests	Penetrationstests
Durchführungszeitpunkt	Implementierung und Testphase	Implementierung	Testphase
Durchführungsressourcen	Arbeitsaufwand: Gering bis Hoch, Zeitaufwand: Gering bis Hoch	Arbeitsaufwand: Hoch bis sehr Hoch, Zeitaufwand: Hoch	Arbeitsaufwand: Hoch bis sehr Hoch, Zeitaufwand: Hoch bis sehr Hoch
Benötigte Durchführungskenntnisse	Geringe Kenntnisse	Gute bis sehr gute Kenntnisse	Sehr gute Kenntnisse
Benötigte Fachkenntnisse im Bereich IT-Sicherheit	Geringe Fachkenntnisse	Gute bis sehr gute Fachkenntnisse	Sehr gute Fachkenntnisse
Fehlerbehebungsressourcen	Kostengünstig; Geringer bis mittlerer Arbeits- und Zeitaufwand	Kostengünstig; Geringer bis mittlerer Arbeits- und Zeitaufwand	Kostenintensiv; Hoher Arbeits- und Zeitaufwand
Fehlererkennungsrate	Hohe Erkennungsrate	Erkennungsrate ist abhängig von der Anzahl der erstellten Tests	Geringe bis hohe Erkennungsrate
Aussagekraft	Anfällig für False Positive Resultate	Anfällig für False Negative Resultate bei schlechter Testabdeckung	Anfällig für False Negative Resultate; Gefundene Schwachstellen stellen reales Risiko dar

erfordert es keinen Aufwand, da die Analyse automatisch im Hintergrund ausgeführt wird), hat zudem den Vorteil, dass der Entwickler nicht aus seinen Arbeitsprozessen geworfen wird.

2.5 Faktor Mensch

Die Sicherheit von Daten in Anwendungen unterliegt häufig der Nachlässigkeit des Menschen. In vielen Fällen tragen die Nutzer der Anwendung selbst dazu bei, dass ihre Daten missbraucht werden können, indem sie beispielsweise ein schwaches Passwort oder eine unverschlüsselte Internetverbindung verwenden. Der Benutzter ist jedoch nicht der einzige Mensch, der zur Sicherheit der Anwendung beiträgt. Maßgebliche Anteilnahme haben auch diejenigen, die die Software herstellen. Unterläuft den Programmierern ein Fehler bei der Implementierung einer sicherheitsrelevanten Funktion, wie beispielsweise dem Loginprozess, kann auch ein sicheres Passwort den Schaden kaum abhalten.

Schwachstellen können auf verschiedenste Wege in die Software gelangen. Um einige dieser Möglichkeiten aufzuzeigen, entwickelten Tsipenyuk, Chess und McGraw, 2005 eine Taxonomie, die für die Software sicherheitskritische Fehlerkategorien benennt. Die nachstehende Auflistung erläutert die zwei wichtigsten Punkte dieser Taxonomie.

- **Eingabevalidierung und -repräsentation:** Viele Schwachstellen entstehen, weil Eingabewerte und Parameter nicht ausreichend überprüft werden und somit beispielsweise zu Pufferüberläufen führen können.

- **Missbrauch von APIs:** Dieser Punkt besagt, dass viele Fehler unter anderem dadurch entstehen, dass sich die Entwickler nicht an die Vorgaben der benutzten APIs halten. Die Szenarien für eine solche fehlerhafte Benutzung sind vielfältig. In ihrem Artikel führen Tsipenyuk et al. einige Beispiele auf. Darunter fällt auch, dass die Verwender einer API falsche Annahmen über das Verhalten oder Rückgabewerte einer Funktion aufstellen, die in der Dokumentation der API so nicht beschrieben sind. In den objektorientierten Programmiersprachen bieten APIs zudem häufig abstrakte Klassen an, die vom Verwender implementiert werden müssen. Die Autoren sehen hier das Problem, dass sich die Entwickler dabei nicht an die Spezifikationen halten (Tsipenyuk et al., 2005). Im konkreten Beispiel der .NET Krypto-API existiert die abstrakte Klasse `Aes`. Ein Anwendungsentwickler könnte nun von dieser Klasse erben und seine eigene Implementation einfügen, die jedoch nicht den Kriterien des Verschlüsselungsverfahren entspricht. Nach außen hin ist dieses Vergehen jedoch nicht unmittelbar erkennbar, da die Vererbung der `Aes`-Klasse auch eine korrekte AES-Implementierung impliziert.

In beiden dieser zwei Fehlerkategorien kann der Faktor Mensch als Ursache verstanden werden. Sie entstehen durch „Unwissenheit, Leichtsinn oder Flüchtigkeit“ (Frauenhofer, 2013, S. 12). Im ersten Punkt der Taxonomie schenken die Programmierer den Eingabeparametern zu viel Vertrauen bezüglich der Korrektheit (Tsipenyuk et al., 2005, S. 81). Für diese Arbeit ist dabei jedoch besonders interessant, dass die Autoren den Punkt „Missbrauch von APIs“ als den zweitwichtigsten einstufen. Die hier

aufgeführten Beispiele zeigen, auf welche Weise durch den Faktor Mensch Schwachstellen bei der Verwendung von Krypto-APIs in die Software geraten können. Eine genauere Analyse, warum sich Entwickler schwer tun die Verträge einer API einzuhalten, erfolgt in Abschnitt 3.2.

Automatisierte Softwaretestverfahren, darunter auch die statische Codeanalyse, können bei richtiger Verwendung einige dieser Fehler, wie die fehlende Eingabeüberprüfung, effektiv und automatisch identifizieren. Der Faktor Mensch wird daher als Schwachstelle minimiert. Forschungen haben jedoch gezeigt, dass die Entwickler ihre Testwerkzeuge nicht in einem geeignetem Maße beherrschen oder benutzen. Als Resultat nehmen die gefundenen Fehler ab oder die benötigte Testzeit steigt (Aniche & Gerosa, 2010; Daka & Fraser, 2014; George & Williams, 2004; Uwano et al., 2006).

Die Benutzung von Krypto-APIs bei der Softwareherstellung offenbart zudem weitere Schwierigkeiten bei der Fehleridentifikation mit Hilfe von Testwerkzeugen, wie der statischen Codeanalyse. Ein SCAT kann ohne nähre Kontextinformationen die Semantik einer API in Bezug auf die Sicherheit der Software nicht abschließend bewerten. Hier ist ein Entwickler erforderlich, der das nötige Fachwissen besitzt, um diese Bewertung durchzuführen. Der Faktor Mensch wird dadurch in den Testprozessen verstärkt. Ansätze und Prototypen zur Minimierung des Faktor Mensch beim Umgang mit APIs existieren erst seit kurzem (Krüger et al., 2017) und sind daher in den Softwareentwicklungsprozessen kaum etabliert oder nur unter bestimmten Bedingungen (z.B. das Verwenden der Programmiersprache Java) einsetzbar.

Kapitel 3

Forschungsstand

Während in Kapitel 2 wichtige Begriffe beschrieben wurden, analysiert dieses Kapitel die für diese Arbeit bedeutenden Themen anhand der Fachliteratur. Ein besonderer Fokus liegt dabei auf die Ein- und Auswirkungen, die durch menschliches Handeln beim Umgang mit Krypto-APIs und SCATs entstehen. Der Mensch ist in diesem Kontext als Entwickler bzw. Programmierer einer sicherheitsrelevanten Anwendung zu verstehen. Der Abschnitt 3.4 synthetisiert die gewonnenen Erkenntnisse und ordnet sie unter dem Gesichtspunkt dieser Arbeit ein. Abschließend stellt dieses Kapitel das bereits erwähnte SCAT CogniCrypt vor und erarbeitet, basierend auf den errungenen Erkenntnissen, die relevanten Forschungsfragen für diese Arbeit.

3.1 Anforderungen an Krypto-APIs

Krypto-APIs kapseln und abstrahieren Lösungen, die für die verschiedensten Anwendungen und Einsatzgebiete verwendet werden. Um ein Sicherheitsrisiko zu minimieren, müssen sowohl bei der Entwicklung dieser APIs, als auch bei ihrem Umgang besondere Anforderungen erfüllt werden. Aus Forschungen um die Themen APIs und insbesondere Krypto-APIs sind die folgenden Anforderungen zu entnehmen:

1. **Sicheres Konzept:** Diese Anforderung sieht vor, dass die mathematischen und technischen Grundlagen hinter einer kryptographischen Funktion korrekt erarbeitet werden müssen (Krüger et al., 2017).

2. **Korrekte Implementierung:** Dieser Punkt sieht vor, dass die erarbeiteten Konzepte der ersten Anforderung korrekt und fehlerfrei in Softwarecode geschrieben werden. Anderson, 1994 stellte fest, dass die Umsetzung der mathematischen Konzepte in Code eine der Hauptursachen war, warum Systeme Verwundbarkeiten aufwiesen.

3. **Aktualität:** Krypto-APIs, die vom Hersteller nicht mehr gewartet und weiterentwickelt werden, bergen ein hohes Risiko, da fehlerhafter Code unter Umständen nicht mehr korrigiert werden kann. G. Jones und Prieto-Diaz, 1988 empfehlen daher eine Instanz im Entwicklungsprozess, die den Einsatz von wiederverwendbaren Softwarekomponenten organisiert.

4. **Sichere Anwendung:** Die für diese Arbeit wichtige Anforderung berücksichtigt auch den Faktor Mensch bei der Verwendung einer Krypto-API. Verwendet

der Programmierer diese falsch, indem er beispielsweise falsche Parameter oder Konfigurationen benutzt, können Schwachstellen in der Anwendung entstehen (Krüger et al., 2017). Der folgende Abschnitt 3.2 befasst sich detaillierter mit diesem Thema.

3.2 Benutzung von Krypto-APIs

Neuere Untersuchen zeigen, dass die Ursache vieler Schwachstellen mittlerweile bei der Verwendung von APIs liegt (Lazar, Chen, Wang & Zeldovich, 2014; Robillard & DeLine, 2011). Je nach API ist der Umfang an angebotenen Funktionalitäten recht hoch. Zudem etablieren APIs häufig neue Datenstrukturen (Klassen in objektorientierten Sprachen), die der Entwickler nicht kennt und unter Umständen nicht versteht, aber dennoch benutzen muss. Genau diese Komplexität und Vielfalt, die innerhalb einer API herrschen können, sind eine der Ursachen, warum Entwickler unsichere Programme schreiben. Ein Experiment von Acar et al., 2017 konnte zeigen, dass eine gute Gebrauchstauglichkeit von Krypto-APIs ein entscheidender Faktor für die Sicherheit der programmierten Anwendungen ist.

In der Untersuchung von Nadi, Krüger, Mezini und Bodden, 2016 hatten 65% aller befragten Java-Entwickler Schwierigkeiten, bestimmte Funktionen einer Krypto-API auszuwählen. Einen Grund, warum die Entwickler sich schwer tun, sich in Krypto-APIs zurechtzufinden, benennen Nadi et al., weil viele Entwickler die verwendeten Krypto-APIs als zu algorithmisch beschrieben. Die befragten Programmierer äußerten den Wunsch nach einer aufgabenorientierten Verwendung der APIs, sei es durch angepasste Schnittstellen oder bessere Dokumentation. Auch Robillard und DeLine, 2011 oder Acar et al., 2017 konnten feststellen, dass viele Programmierer Probleme haben, das Funktionsangebot einer API auf ihren Anwendungskontext zuzuordnen, da ihnen aufgabenbezogene Programmierschnittstellen fehlten. Als Folge daraus konnte gezeigt werden, dass bis zu 95% aller Android Apps, die in weiteren Untersuchungen analysiert wurden, Krypto-APIs falsch verwenden (Krüger et al., 2018; Chatzikonstantinou, Ntantogian, Karopoulos & Xenakis, 2016; Egele et al., 2013).

Ebenfalls fand man heraus, dass die Entwickler häufig Problemlösungen wählen oder annehmen, die einfach zu programmieren sind, aber beispielsweise veraltete und unsichere Funktionalitäten verwenden (Meng et al., 2018). In dem Beispiel, das Meng et al. nennen, versucht ein Entwickler einen selbst signierenden Server mit einem Webdienst über HTTPS zu verbinden. Da sein Verbindungsaufbau jedoch nicht gelingen will, schlägt er selbst vor, die Verifizierung zu umgehen und fragt nach Hilfe, welche er auch bekommt und anwendet[1]. Der Entwickler hat also eine einfache Lösung akzeptiert und dadurch seine Anwendung unsicherer gemacht. Dieses Beispiel zeigt, dass die Entwickler einige allgemeine Kenntnisse in Bezug auf Sicherheitsprinzipien besitzen. So wissen sie beispielsweise was SSL und HTTPS sind und können auch feststellen, dass diese Technologien für ihre Probleme bei der Entwicklung ihrer Anwendung verantwortlich sind. Über Schwächen spezieller Aspekte, darunter auch die Konfigurationsmöglichkeiten von symmetrischen Verschlüsselungsverfahren, so zeigen

[1] https://stackoverflow.com/questions/21156929/java-class-to-trust-all-for-sending-file-to-https-web-service

Nutzerfragen[2], haben sie jedoch keine Kenntnis. Des Weiteren zeigt das Beispiel, dass die Entwickler nicht in der Lage sind, sich mit diesen Themen im Kontext ihrer Anwendung auseinanderzusetzen. Der Entwickler schilderte, dass er bereits sämtliche Anleitungen ausprobiert hatte, bevor er auf Stackoverflow.com nach Hilfe fragte. Die Erkenntnis, dass Entwickler gerne eine angenehme Lösung ihrer Probleme wählen, lässt sich auch auf die Benutzung von Krypto-APIs anwenden. Dazu folgendes Gedankenbeispiel:

> Angenommen ein Entwickler wolle Passwörter sicher in Form eines Hashwertes abspeichern. Er benutzt dazu eine API, von der er gelesen hat, dass sie sichere Hashingmethoden bereitstellt. Er verwendet diese API jetzt das erste mal und kennt sich daher mit ihrer Struktur und ihrem genauen Funktionsangebot nicht aus. Ein schnelles durchforsten der API Dokumentation ergibt, dass mehrere Hashingfunktion verfügbar sind. Die Eingabe- und Rückgabewerte unterscheiden sich jedoch in ihrer Menge und ihrem Typ (gemeint ist der Typ eines Objekts in einer objektorientierten Sprache). Von den verschiedenen Hashfunktionen besitzt MD5 den Methodenkopf (z.B.: `string HashMD5(string input);`), der am einfachen zu verwenden ist.

In vielen Fällen, kann nun nach der Aussage von Meng et al. davon ausgegangen werden, dass sich der Entwickler mit dem MD5-Verfahren zufrieden gibt, da es am schnellsten und einfachsten benutzt werden kann. Wang, Feng, Lai und Yu, 2004 zeigen jedoch, dass MD5 für Kollisionsangriffe anfällig ist und daher als Passwortschutz ungeeignet ist. Der Entwickler hat also wegen seiner Unkenntnis über die API und dem Drang angenehm programmieren zu können, eine angreifbare Software geschrieben.

Die oben aufgeführten Daten und Berichte zeigen deutlich, dass der Ursprung von Schwachstellen in Software oftmals in der Entwicklungsphase liegt. Die hier geschilderten Hauptgründe für diese fehlerhafte Verwendung werden im Folgenden kurz zusammengefasst:

Entwickler

- empfinden die Gebrauchstauglichkeit der APIs aufgrund ihres abstrakten und nicht aufgabenorientierten Konzepts als schlecht.
- sind nicht in der Lage spezielle Aspekte kryptographischer Verfahren differenziert zu bewerten.
- sind mit dem Funktionsangebot einer API überfordert.
- verlassen sich auf schnelle, einfache und womöglich falsche Problemlösungen.
- verstehen das Design, Struktur oder Dokumentation einer API nicht.

Die Meisten dieser Gründe legen eine schlechte Dokumentation der API nahe. Allgemeine Untersuchungen fanden heraus, dass Codebeispiele den Entwicklern helfen

[2]https://crypto.stackexchange.com/q/967, Zugriff: 29.12.2018

Tabelle 3.1: Kryptographische Aufgaben und ihre Verteilung in 100 Github-Projekten nach Nadi, Krüger, Mezini und Bodden, 2016

Aufgabe	**Anteil in allen Projekten**
Symmetrische Verschlüsselung	64%
Signierung und Verifikation von Daten	42%
Generierung geheimer Schlüssel	23%

eine API zu benutzen (McLellan, Roesler, Tempest & Spinuzzi, 1998; Nykaza et al., 2002). Es muss jedoch differenziert betrachtet werden, welchen Umfang diese Codebeispiele besitzen sollen. Zu kleine Beispiele haben das Problem, nicht die Probleme abzudecken, die der Entwickler erklärt haben möchten. Zu lange Beispiele hingegen sind entweder zu komplex, um sie zu verstehen oder werden ohne weiteres Überlegen kopiert und in die Anwendung integriert (Robillard & DeLine, 2011, S. 718). Ferner ging aus einem Entwicklerinterview hervor, dass die Dokumentation einen starken Bezug auf den Anwendungskontext haben muss. Neue Datenstrukturen und Funktionen in APIs können auf diese Weise den Entwicklern einfacher nahegebracht werden (Robillard & DeLine, 2011, S. 720).

Um nun anwendungsbezogene Lösungen für diese Probleme vorzuschlagen, lohnt sich ein Blick auf Statistiken, die zeigen, welche Art von kryptographischen Aufgaben Entwickler am häufigsten verwenden. Beim Entwerfen solcher Lösungen können diese Statistiken dann als Argumentationsbasis dienen, welche Ziel- und Aufgabengruppe priorisiert werden soll. Beispielsweise untersuchten Nadi et al. die Verteilung von Krypto-Aufgaben innerhalb von 100 Github-Projekten[3]-Projekten (Nadi et al., 2016). Als Krypto-Aufgabe, bzw. kryptographische Aufgabe, kann eine Aufgabe verstanden werden, deren Umsetzung die Benutzung einer Krypto-API erfordert. Die Tabelle 3.1 zeigt diese Verteilung. Daraus geht hervor, dass die symmetrische Verschlüsselung am relevantesten ist. Das Generieren geheimer Schlüssel kommt bei ca. einem Viertel aller Projekte vor. Es sei erwähnt, dass pro Projekt mehre Aufgaben identifiziert werden können, weshalb die Summe der prozentualen Anteile der Tabelle nicht 100% ergeben muss. Anhand dieser Statistik können Prioritäten abgeleitet werden, die als Argumentationsbasis dienen in welche Richtung sich weitere Forschungen konzentrieren sollten.

3.3 Statische Codeanalyseprogramme

Die folgenden Abschnitte führen wichtige Erkenntnisse auf, die Forschungen in Bezug auf statische Codeanalyseprogramme erbracht haben. Besonderer Fokus liegt dabei auf der Interaktion dieser Programme mit den Entwicklern, welchen sie bei ihrer Arbeit unterstützen sollen.

3.3.1 In der Softwareentwicklung

Der Einsatz von SCATs in der Softwareentwicklung kann durch zwei primäre Faktoren begründet werden. Durch die Verwendung dieses Codereviewwerkzeugs kann die

[3]http://www.github.com

Qualität und damit auch die Sicherheit der Software gesteigert werden (Nagappan & Ball, 2005; J. Viega et al., 2000). Ihr Einsatz bietet sich zudem in der frühen Entwicklungsphase einer Software an, da hier von 30% (Baca, Carlsson & Lundberg, 2008, S. 86) bis über 50% (Chelf & Ebert, 2009, S. 97) der Fehler gefunden werden können. Besonders für Unternehmen spielt auch der zweite Faktor eine ausschlaggebende Rolle. So konnte gezeigt werden, dass SCATs die Entwicklungskosten bis zu 30% senken können (Baca et al., 2008, S. 87; Chelf & Ebert, 2009, S. 97). Softwarefehler, die während der Entwicklungsphase behoben werden, sind zudem um ein Vielfaches günstiger, als solche, die erst nach dem Veröffentlichen der Software behoben werden (Tassey, 2002). Daher kann man argumentieren, dass von einem wirtschaftlich gesehenen Standpunk, SCATs eines der wirkungsvollsten Werkzeuge sind (Kleidermacher, 2008, S. 367). An dieser Stelle muss jedoch auch erwähnt werden, dass SCATs nicht alle Fehler finden und daher andere qualitätssichernde Schritte nicht ersetzen (Chelf & Ebert, 2009, S 96). Ebenfalls wurde herausgefunden, dass die Fehlererkennungsrate von SCATs hinter ihren Erwartungen liegt, sie jedoch zur Weiterbildung und zum Wissensaustausch bei den Entwicklern führen (Bacchelli & Bird, 2013). Da SCATs selbst Programme sind, bringen sie zudem den Vorteil mit, dass sie ihre Aufgabe wesentlich schneller und häufiger durchführen können als Menschen, die den Code manuell bewerten müssen (Novak et al., 2010).

Derzeit gibt es zwei weit verbreite Arten, in welchen SCATs genutzt werden. Man unterscheidet hierbei zwischen einem intrigierten und externen SCAT. Unter einem externen SCAT versteht man, dass das Tool als eigenständiges Programm neben den anderen Editoren, IDEs und weiteren Tools ausgeführt wird. Bei der integrierten Form hingegen, wird das Tool in die bereits vorhandenen Entwicklungswerkzeuge integriert. Übliche Arten dieser Bereitstellung ist das Installieren eines Plugins für die IDE. Das Plugin erweitert dann die IDE um die Funktionen des SCAT. Bei dieser Form von Tools existiert der Vorteil, dass der Entwicklungsprozess nicht dahingehend erweitert werden muss, dass die Entwickler oder ein Qualitätssicherungsteam geschriebenen Code noch einmal, möglicherweise mit viel Aufwand verbunden, durch ein weiteres Tool analysieren lassen müssen. Der Code wird bereits in der vertrauten Umgebung des Entwicklers analysiert (Chelf & Ebert, 2009, S. 97) und Fehler können somit bereits früh in der Entwicklungsphase entdeckt und verbessert werden (Baca et al., 2009, S. 804). Eine Untersuchung ergab, dass diese Art der SCATs von Entwicklern gegenüber den externen Tools bevorzugt wird (B. Johnson, Song, Murphy-Hill & Bowdidge, 2013, S. 680).

Standardmäßig sind SCATs jedoch nicht in der Lage anwendungsspezifische Fehler zu erkennen. Daher bieten sie zudem die Möglichkeit an, neue Erkennungsklassen oder Analysemuster, ihrem aktuellem Regelwerk hinzuzufügen. Ein derartiges Anpassen eines SCAT erfordert jedoch einen gewissen Konfigurationsaufwand und setzt eine Expertise desjenigen voraus, der diese Konfiguration durchführt (Ersoy & Sözer, 2016). Zudem erlaubt diese Erweiterbarkeit das Updaten und Instandhalten der Tools. Alte bzw. veraltete Analyseprogramme sind nach der Meinung von Novak et al., 2010, S. 419 nicht für den Produktiveinsatz tauglich.

Ein wesentlicher Unterschied zwischen einer statischen und einer dynamischen Codeanalyse ist, dass bei Ersterem der Code nicht ausgeführt werden muss. Die Analyse ist nicht auf dynamische Eingabewerte angewiesen und kann daher weniger umständlich Fehler erkennen (Chess & McGraw, 2004, S. 76). An vielen Stellen wird jedoch ausdrücklich betont, dass der Einsatz einer statischen Codeanalyse nie-

mals alle Fehler und Schwachstellen finden kann und daher weitere Maßnahmen zur Herstellung von sicherer Software nicht ersetzt (Chess & West, 2007, S. 56; Chelf & Ebert, 2009). Baca et al., 2008 haben in ihrem Experiment herausgefunden, dass besonders anwendungsspezifische Fehler nicht von SCATs erkannt wurden. Ebenso hätten viel mehr Programmierfehler aufgedeckt werden müssen. In einem anderen Versuch konnte festgestellt werden, dass selbst nach dem Kombinieren der acht untersuchten Tools weniger als 50% der möglichen Fehler entdeckt wurden (Ware & Fox, 2008, S. 16).

Gerade anwendungsspezifische Fehler, zu denen besonders das Design der Software und die Programmiersprache zählen, werden selten erkannt, da viele SCATs eher dafür ausgelegt sind, Fehler zu erkennen, die am häufigsten auftreten (Sun, Shu, Podgurski & Robinson, 2012). Besonders SCATs, die auf einem festen Regelwerk basieren, sind von dieser Schwäche stark betroffen, da diese Tools nicht in der Lage sind Fehler zu finden, für die keine Regel existiert (Novak et al., 2010; Chess & McGraw, 2004). Um diesen Nachteil auszugleichen, können SCATs konfiguriert und ihre Regelwerke erweitert werden. Durch die Anpassung von Einstellungsmöglichkeiten konnte bereits gezeigt werden, dass sowohl mehr Fehler erkannt wurden (Ware & Fox, 2008), als auch weniger (Baca et al., 2008).

Es sind jedoch auch Einschränkungen, besonders bei der Interaktion zwischen Entwickler und SCAT, beobachtbar, die dazu führen, dass SCATs nicht in dem Umfang eingesetzt werden, wie man erwarten könnte. Die Ursachen dafür werden im folgenden Abschnitt genauer erläutert.

3.3.2 Verwendung statischer Codeanalyseprogramme

SCATs sind keine vollständig autonomen Werkzeuge und bedürfen daher menschlicher Bedienung (Chess & McGraw, 2004). Ihr Einsatz während der Implementationsphase legt nahe, dass die Softwareprogrammierer den meisten Kontakt mit diesen Tools haben. Welchen Einfluss SCATs auf Entwickler haben, welche Aspekte beachtet werden müssen und wo Schwächen bei der Zusammenarbeit zwischen Mensch und Werkzeug auftreten, betrachtet dieser Abschnitt.

Baca et al., 2009 konnten einerseits feststellen, dass Entwickler trotz der Nutzung von SCAT, Schwachstellen nicht besser einschätzen konnten, betonten jedoch auf der anderen Seite die Chance, dass SCATs durchaus zur Erkennung von Schwachstellen beitragen können. Als ein wesentlicher Hauptfaktor, der erfüllt sein muss, benennen sie die Nutzererfahrung mit dem entsprechenden Tool. Ist diese hoch, arbeiten die Entwickler effizienter (Baca et al., 2009, S. 809). Im Wesentlichen kommuniziert ein SCAT über Fehler- und Statusberichte mit dem Entwickler. In Folge dessen, müssen diese Mitteilungen dem Programmierer klar verständlich sein. Wird ein Fehler gefunden, muss dieser dem Entwickler auf eine Art mitgeteilt werden, dass direkt erkennbar ist, wo der Fehler liegt. Der Entwickler sollte nicht zusätzlichen Aufwand betreiben, den Fehler im Code zu suchen (Baca et al., 2009).

Die Soundness (dt.: Korrektheit) eines SCAT informiert über die Erkennungsrate dieser Tools. Ist ein Analyseprogramm sound, so existieren keine sogenannten *False Negative* Treffer. False Negatives sind Softwarefehler im Code, die fälschlicherweise nicht von der Analyse erkannt wurden. Traditionell gesehen bedeutet Soundness zudem, dass das Tool keine *False Positives* erzeugt. Ein False Positive Treffer ist als falscher Alarm zu verstehen. Das Tool meldet einen Fehler oder Schwachstelle, obwohl

an der identifizierten Stelle dieses Merkmal nicht vorliegt (Emanuelsson & Nilsson, 2008, S. 6). Berichtet ein SCAT keine False Positive Meldungen, spricht man heutzutage eher von einem vollständigem Tool (Hicks, 2017). Nach dem Satz von Rice (Rice, 1953) können SCATs jedoch nicht sowohl sound als auch vollständig sein. SCATs stellen daher immer nur eine Approximation dar. Ist das Tool sound, so wird die Analyse überschätzt, sodass mehr Treffer entstehen können. Ist das Tool vollständig wird die Analyse unterschätzt, sodass Treffer fehlen können.

In Untersuchungen wurde beobachtet, welche Wirkung diese Klassifikationen auf einen Anwender haben kann. Besonders False Positives können die Nutzererfahrung stark beeinflussen (B. Johnson et al., 2013). Eine höhere Rate von False Positives bedeutet auch, dass das Tool mehr Treffer berichtet. Der Entwickler steht nun in der Verantwortung, sich alle Treffer anzugucken und muss dann entsprechend bewerten, ob es sich um einen richtig erkannten Fehler oder um einen falschen Alarm handelt. Shen, Fang und Zhao, 2011 haben herausgefunden, dass die Entwickler bei einer hohen False Positive Rate viel mehr Zeit benötigen. Je nach zu überprüfender Code- und Fehlermenge haben die Entwickler jedoch nicht immer die Zeit, von allen gefundenen Meldungen die richtigen herauszusuchen (Bessey et al., 2010, S. 69). Zusätzlich zu diesem Problem wurde herausgefunden, dass Entwickler häufig Schwierigkeiten haben, die gefundenen Meldungen richtig einzuordnen. Die Programmierer neigen dazu, den Meldungen des Analysetools zu sehr zu vertrauen oder zeigen ein falsches Selbstbewusstsein, was die Bewertung der Meldungen angeht (Baca et al., 2009). Des Weiteren kann eine sehr hohe False Positive Rate negativen Einfluss auf die Arbeitsmoral des Entwicklers haben, da er einerseits viele Meldungen überprüfen muss und andererseits erwartet, seinen eigenen Code oder den anderer ändern zu müssen (Chess & McGraw, 2004). Beim Anpassen der False Positive Rate ist jedoch Vorsicht geboten, da das Risiko besteht, die Rate der False Negatives zu steigern (Emanuelsson & Nilsson, 2008, S. 5; Huang et al., 2004, S. 44).

Während sich False Positives überwiegend negativ auf die Benutzererfahrung des SCAT auswirken, haben False Negatives einen unmittelbaren Einfluss auf die Sicherheit der Software. Sie geben dem Entwickler ein falsches Gefühl der Sicherheit, da das SCAT keine Fehler meldet (Chess & McGraw, 2004) und die Entwickler, wie oben erwähnt, den Tools viel Vertrauen entgegenbringen. Das Finden solcher False Negatives ist jedoch nicht leicht (Zheng et al., 2006). SCATs mit einer hohen False Positive Rate sind daher eher für Sicherheitsexperten in der Testphase der Software zu empfehlen, die False Positives von *True Positives* unterscheiden können (Braga, Dahab, Antunes, Laranjeiro & Vieira, 2017, S. 179).

Das hohe Vertrauen der Entwickler in ihre Tools wirkt sich jedoch positiv in der Bewertung von True Positives aus. So wurde festgestellt, dass sich die meisten Entwickler bei ihrer Fehlerbewertung sehr sicher waren, dass die gefundene Meldung auch wirklich eine Schwachstelle darstellt (Baca et al., 2009).

Des Weiteren wird die Qualität der Berichte vieler SCATs kritisiert. So mangele es beispielsweise oft an Hinweisen und Unterstützungen, die den Entwickler wissen lassen, worum es in dem konkreten Fehler geht und wie er behoben werden kann. Wurde dieser korrigiert, muss der Eintrag vom SCAT sofort aus dem Fehlerbereich entfernt werden, da sonst Verwirrung beim Entwickler entstehen kann (Ayewah, Hovemeyer, Morgenthaler, Penix & Pugh, 2008, S. 25).

Viele SCATs tendieren aus den hier erläuterten Gründen eher zur Vollständigkeit als zur Soundness (Emanuelsson & Nilsson, 2008), da die Nutzererfahrung sonst als

zu schlecht empfunden wird. Baca et al., 2008 ermittelten in ihrer Fallstudie beispielsweise eine False Positive Rate von 5-22%, welche sie als akzeptables Ergebnis einstuften. Bei False Negatives ist es wichtig die Rate so gering wie möglich zu halten, da nicht gefundene Fehler direkte Auswirkungen auf die Sicherheit der Software haben können. Schon Fehlerraten von 2-10% werden als klare Bedrohung angesehen (Cova, Kruegel & Vigna, 2010).

B. Johnson et al., 2013, S. 677 zitieren in ihrer Untersuchung eine Entwicklerin, die anmerkt, dass eine Software, die erkennen kann, ob ein Fehler vorliegt, auch in der Lage sein sollte, eine Korrektur vorzuschlagen. Moderne SCATs sind bereits in der Lage sogenannte *Quick-Fixes* für einige Fehler vorzuschlagen und können diese zum Teil auch direkt in den bestehenden Code integrieren. Befragungen von Entwicklern brachten zur Kenntnis, dass Quick-Fixes erst nach vorherigem Review der Entwickler angewendet werden sollten. Ebenfalls wurde ersichtlich, dass Quick-Fixes eine Funktion darstellen, die Zeit und Aufwand einsparen sowie SCATs gebrauchstauglicher machen können (B. Johnson et al., 2013, S. 680). Ferner geben B. Johnson et al., 2013 jedoch noch zu bedenken, dass Quick-Fixes für komplizierte Lösungen unpraktikabel sind und möglicherweise dazu führen, dass die Entwickler weniger über ihren Code nachdenken und stattdessen den vorgeschlagenen Quick-Fix anwenden. Diese Aussage deckt sich auch mit der bereits oben vorgestellten Erkenntnis von Meng et al., 2018, nach der Entwickler gerne zu Lösungen tendieren, die schnell und einfach anwendbar sind.

Abschließend soll darauf aufmerksam gemacht werden, dass das Nutzungsverhalten und Funktionsangebot und die daraus folgende Nutzererfahrung mit einem SCAT stark abhängig vom Entwickler bzw. der nutzenden Zielgruppe ist. Baca et al., 2009, S. 809 erklären zum Beispiel, dass die Soundness bei einem Qualitätssicherungsteam wichtiger wäre, als bei einem Entwickler der Anwendungscode schreibt.

3.3.3 Vorteile und Verbesserungsmöglichkeiten von SCATs

Aus den hier bereits vorgestellten Erkenntnissen und Aussagen lässt sich eine Sammlung von Attributen festhalten, welche die Vorteile und Verbesserungsmöglichkeiten von SCATs beschreiben.

Vorteile

- **Erkennung von Fehlern und Schwachstellen**. SCATs sind erwiesenermaßen in der Lage, Probleme und Fehler in einer Software zu finden. Sie eigenen sich daher als Werkzeug für die Qualitätssicherung und Erstellung sicherer Software.

- **Bereitstellung von Feedback über den Quellcode**. Mit Hilfe des Feedbacks, können Entwickler über Fehler und mögliche Schwachstellen in der Software aufgeklärt werden. Zusätzlich erspart es dem Entwickler Arbeit, da dieser den Fehler nicht selbst im Quellcode identifizieren muss, sondern ihn anhand des Feedbacks ablesen kann.

- **Integration in den Entwicklungsprozess**. SCATs sind nicht selten Programme, die als Plugin einer IDE verfügbar sind und mit dieser direkt intera-

gieren. Der Entwickler wird somit nicht aus seinem vertrauten Arbeitsprozessen herausgerissen und kann mit dem SCAT kohärent arbeiten.

- **Einfachheit**. Untersuchungen haben gezeigt, dass SCATs auch ohne zusätzliche Konfiguration in der Lage sind, Software erheblich sicherer zu gestalten. Da die Konfiguration eines Tools oft einen Mehraufwand bedeutet und zudem ein Verständnis dieses Tools voraussetzt, ist eine Benutzung ohne Konfiguration für den Arbeitsprozess förderlich.

- **Analysefrequenz**. SCATs haben den Vorteil, dass sie den Code nach jeder Codeeingabe des Entwicklers erneut untersuchen können. Der Entwickler erhält also im Optimalfall unmittelbar nach seiner Eingabe ein Feedback.

- **Analysegeschwindigkeit**. Ein Tool kann Code sehr viel schneller untersuchen als ein Mensch. Ebenfalls bleibt die Erkennungsrate immer konstant und ist nicht an menschliche Faktoren wie Konzentration gebunden. Zudem kann ein Tool den Code parallel zur Eingabe untersuchen, was schnellere Resultate hervorbringt, als den Code erst nach jedem Übersetzungsvorgang zu untersuchen.

- **Präzision**. Durch Regeln oder anderen Technologien sind SCATs in der Lage, Wissen über bestimmte Aspekte der Softwareentwicklung wie Verwundbarkeiten zu bündeln und dieses Wissen dann auf die Codeanalyse anzuwenden. Untersuchungen ergaben auch, dass sich die Ergebnisse der Codeanalysetools im Laufe der Zeit verbessert haben.

- **Erweiterbarkeit**. Viele SCATs können durch entsprechende Konfiguration erweitert werden. Dadurch sind sie in der Lage, den Code nach anwendungsspezifischen Kriterien zu untersuchen.

Verbesserungsmöglichkeiten

- **Soundness und Vollständigkeit**. Obwohl SCATs bereits viele Erkennungsklassen bei ihrer Analyse unterstützen, haben viele dieser Tools noch das Problem den Kontext einer Anwendung nicht verstehen zu können. Das Resultat ist, dass die Tools entweder keine Aussage treffen oder False Positive Treffer entstehen. Des Weiteren können SCATs erwiesenermaßen niemals alle Fehler in einer Software finden, sondern immer nur eine mehr oder weniger gute Approximation liefern. Je nach Anwendungskontext der Benutzung eines SCAT, spielen eine gute Soundness oder Vollständigkeit eine andere Rolle.

- **Zu statisch**. Dieser Punkt sagt aus, dass SCATs derzeit noch größtenteils der Limitation unterstehen, nur Fehler finden zu können, die ihnen, beispielsweise durch Regeln, beigebracht wurden.

- **User Experience**. Die Nutzbarkeit eines Tools scheint für dessen Erfolg von elementarer Bedeutung zu sein. Eine schlechte Nutzererfahrung mindert das Resultat der Arbeit mit diesem Tool stark und kann sogar dazu führen, dass das Tool nicht verwendet wird. Auf der Gegenseite jedoch vertrauen viele Entwicklern ihren Tools zu sehr und lassen sich durch False Positves in die Irre führen oder haben im Falle von False Negatives ein falsches Gefühl der Sicherheit.

3.4 Zusammenfassung des Forschungsstands

Die bisher aufgeführten Untersuchungen machen vor allem deutlich, dass die Entwickler maßgeblich für die Sicherheit ihrer Anwendung verantwortlich sind. Als relevante Faktoren, warum Entwickler unsicheren Code produzieren, wurden einerseits ein Mangel an Fachwissen sowie die Bequemlichkeit des Menschen identifiziert. Es zeigte sich, dass der Mangel an Fachwissen mit zunehmender Spezialisierung steigt. Waren die meisten Programmierer noch in der Lage diverse kryptographische Verfahren differenziert zu beschreiben (abstraktes, allgemeines Fachwissen), viel es ihnen jedoch sehr schwer, die ihnen bekannten Verfahren in Krypto-APIs wiederzufinden oder zu verwenden (spezielles Fachwissen). Der Punkt Bequemlichkeit kam besonders dann zum Tragen, wenn die Programmierer aus Mangel an Erfahrung oder zunehmender Ausweglosigkeit nach Hilfe fragten. Sie waren bereit, die Sicherheit der Anwendung bewusst zu vernachlässigen, um eine schwer zu entwickelnde Funktion fertigzustellen. In diesem Kontext übernahmen die Entwickler auch häufig den ersten unterbreiteten Hilfsvorschlag, ohne diesen genauer zu differenzieren.

Der Einsatz von Testwerkzeugen, wie beispielsweise einem SCAT, brachte nur bis zu einem gewissen Grad einen Vorteil. Auch hier spielen die oben benannten Faktoren eine wichtige Rolle. Es konnte gezeigt werden, dass Entwickler, die mit dem Umgang der Testwerkzeuge nicht vertraut waren, schlechtere Resultate erbrachten. Das fehlende Fachwissen der Programmierer, zeigte sich besonders bei einer hohen False Positive Rate eines SCAT. Der Entwickler wird einerseits mit vielen Warnmeldungen überfordert und andererseits fehlt ihm oft die Erfahrung ein False Positive von einem True Positive zu unterscheiden. Der Faktor Bequemlichkeit hat besonders bei False Negative Fällen eine schwerwiegende Konsequenz. Die Programmierer entwickelten ein falsches Vertrauen in ihre Werkzeuge, sodass sie zu der Annahme kamen, ihr Code enthielte nur die Fehler, die das SCAT auch identifiziert hat. Ähnlich verhielt es sich mit dem Einsatz von Quick-Fixes, die ein SCAT bereitstellen kann. Hier zeigte sich, analog zu den oben erwähnten Hilfsvorschlägen, dass die Entwickler diese Quick-Fixes oftmals undifferenziert anwendeten.

3.5 CogniCrypt als Beispiel eines statischen Codeanalyseprogramms der Nutzung von Krypto-APIs

CogniCrypt[4] hebt sich besonders von den im Abschnitt 2.3 genannten Tools ab, da es sich sehr stark auf eine Aufgabe konzentriert und nicht, wie die anderen, als Allzwecktool ein breites Spektrum von Programmfehlern abdecken will. Dadurch ergeben sich die Vorteile, dass CogniCrypt viel präziser auf die besonderen Anforderungen bei der Verwendung von Krypto-APIs eingehen kann. Es wurde basierend auf den Erkenntnissen von Nadi et al., 2016 entwickelt, die in Abschnitt 3.2 beschrieben wurden. CogniCrypt ist als Open Source Software unter dem in der Fußnote angegeben Link frei verfügbar. Die Zielplattform ist die IDE Eclipse.

Zu seinen Hauptmerkmalen gehören neben einer speziell auf die Java Cryptographic

[4] https://github.com/eclipse-cognicrypt/CogniCrypt, Zugriff: 30.12.2018

Architecture (JCA)-API basierenden Codeanalyse, auch das Generieren von sicherem Code für eine kryptografische Aufgaben, wie beispielsweise dem Verschlüsseln von Daten. Dem Entwickler werden somit zusätzliche Werkzeuge und Informationen auf eine Weise bereitgestellt, die andere SCATs nicht unterstützen. Zu diesen Werkzeugen gehören unter anderem die spezielle Art der Codegenerierung, die mit einer Art Installationsassistent vergleichbar ist. CongiCrypt generiert anhand des Assistenten den aufgabenbezogenen Code und integriert ihn direkt in das Anwendungsprojekt. Derartige Quick-Fixes sind zwar auch in anderen SCATs vorhanden, jedoch ist diese spezielle Aufbereitung recht einzigartig und für den Entwickler auf besondere Weise zugänglich.

3.5.1 Codeanalyse

Das Funktionsangebot der Codeanalyse von CogiCrypt beschränkt sich aktuell auf die JCA. Wird ein Fehler bei der Nutzung der Krypto-API identifiziert, wird eine Fehlermeldung im Fehlerfenster „Problems“ von Eclipse angezeigt. In ihrer Veröffentlichung führen Krüger et al., 2017, S. 932 das Beispiel der Verwendung der Factory-Methode `Cipher.getInstance()` vor. Diese Methode erlaubt das Konfigurieren eines `Cipher`-Objekts durch das Angeben einer Zeichenkette. Andere SCATs analysieren üblicherweise nicht den Inhalt von Zeichenketten, sondern lediglich ihre Länge oder den `null`-Status. In diesem Fall kann, je nach Inhalt der Zeichenkette, die Sicherheit der Daten stark variieren. CogniCrypt überprüft daher auch den Inhalt und kann eine Aussage treffen, ob die angegebene Konfiguration des `Cipher`-Objekts sicher ist oder nicht. Die Abbildung 3.1 zeigt einen fehlerhaften Code und die Problemmeldung durch CogniCrypt. In diesem Beispiel wird der Methode `getInstance()` der Parameter “`AES`“ übergeben. Laut einer Nutzerantwort auf Stackoverflow.com[5] ist dieser Parameter gleichzusetzen mit “`AES/ECB/PKCS5Padding`“, einer unsicheren Konfiguration von AES[6]. Zu erkennen ist auch, dass in der Fehlermeldung ein Lösungsvorschlag gegeben wird, mit welchem diese Schwachstelle behoben werden kann.

Technisch umgesetzt wird die Codeanalyse durch eine eigens entwickelte Regelsprache namens *CrySL* (Krüger et al., 2018). CrySL arbeitet nach einem White-List Verfahren und spezifiziert so konkret, wie die einzelnen Funktionalitäten der JCA zu verwenden sind. Eine Abweichung von dieser Spezifikation resultiert in einer Fehlermeldung durch CogniCrypt.

Auf der Dokumentationsseite von CrySL[7] findet sich neben einer kurzen Syntaxerklärung der Sprache auch das aktuelle Analyseangebot, welches von CogniCrypt unterstützt wird. Darunter gehören auch Analysen zur Konfiguration symmetrischer und asymmetrischer Kodierungsverfahren, die Schlüsselgenerierung oder Hashingverfahren.

[5] https://stackoverflow.com/a/10084030, Zugriff: 07.12.2018
[6] https://zachgrace.com/posts/attacking-ecb/, Zugriff: 07.12.2018
[7] https://www.eclipse.org/cognicrypt/documentation/crysl/, Zugriff: 07.12.2018

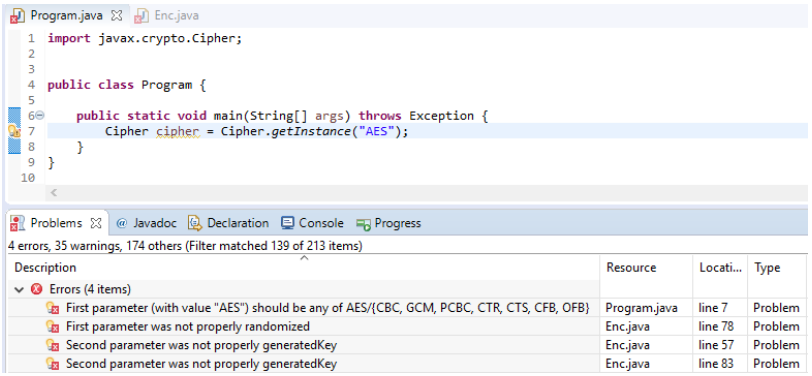

Abbildung 3.1: Fehlerdarstellung einer unsicheren AES-Konfiguration der JCA durch CogniCrypt

3.5.2 Codegenerierung

Wie bereits erläutert, unterstützt CogniCrypt das Generieren von Codefragmenten mit Hilfe eines Assistenten. Wie in Abschnitt 3.2 vorgeschlagen, werden die verfügbaren Codegenerierungen in Krypto-Aufgaben aufgeteilt. Die Namen der Aufgaben sind dabei möglichst sprechend gehalten und versuchen somit den Anwendungsentwickler zu unterstützen. Die aktuell verfügbaren Aufgabenkategorien sind in Abbildung 3.2 einsehbar. Unter dem ersten Symbol kann sowohl die symmetrische, als auch asymmetrische Verschlüsselung gewählt werden. Mit Auswahl des Schlüsselsymbol kann Code zur sicheren Schlüsselgenerierung erzeugt werden. Das WLAN-Symbol steht für generierbaren Code zur Herstellung sicherer Internetverbindungen und über den Doktorhut können experimentelle Algorithmen generiert werden. Deutlich erkennbar ist, dass die am häufigsten gebrauchten Aufgaben, wie sie Tabelle 3.1 aufführt, bereits vorhanden sind. Mit dem Assistent können nun weitere, wichtige Informationen für den neuen Code angegeben werden. Der Entwickler beantwortet dazu einfache, aufgabenbezogene Fragen. Diese werden sprachlich so gestellt, dass Expertenwissen im Bereich der Kryptographie nicht erforderlich ist. Die Abbildung 3.3 zeigt eine Seite des Assistenten zur Codegenerierung einer symmetrischen Verschlüsselung. Mit Abschluss des Assistenten wird der passende Code generiert und in das aktuelle Softwareprojekt eingefügt. CongniCrypt erstellt dazu ein neues Java-Package namens `Crypto`, in dem die generierte Funktionalität implementiert ist. Zusätzlich wird in der aktuell geöffneten Codedatei ein kurzes Beispiel mit dem Methodennamen `templateUsage` generiert, das zeigt, wie der neue Code zu verwenden ist. Die Abbildung 3.4 zeigt die Projektstruktur, nachdem die Codegenerierung abgeschlossen wurde.

3.5.3 Verbesserungsmöglichkeiten

Auch wenn CogniCrypt derzeit sehr neu ist und noch aktiv weiterentwickelt wird, ist eine kritische Auseinandersetzung einiger Funktionen wichtig. Besonders vor dem Hintergrund, dass in dieser Arbeit ein vergleichbares Analyseprogramm entwickelt wird, können somit entscheidende Erkenntnisse erlangt werden, die bei der Entwicklung des Tools direkt umgesetzt werden können. Die wohl interessanteste Erkenntnis ist, dass die Codegenerierung der symmetrischen Verschlüsselung mit Passwortunterstützung von CogniCrypt (Version 1.0.0.201812071312) Code erzeugt, der selbst

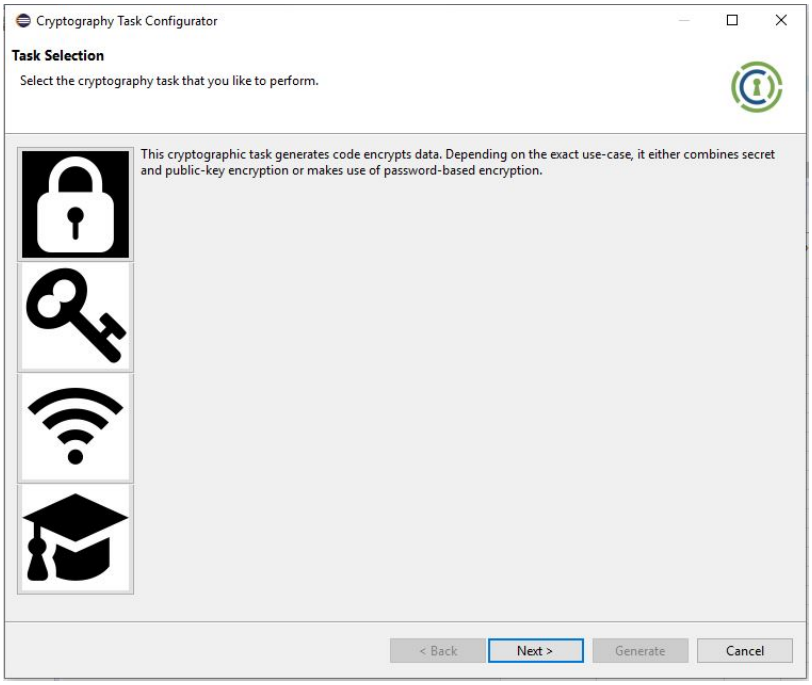

Abbildung 3.2: Verfügbare Krypto-Aufgaben für die Codegenerierung in CogniCrypt

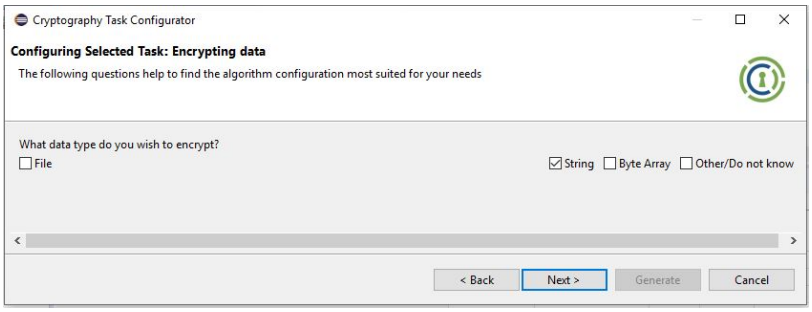

Abbildung 3.3: Codegenerierungsassistent für eine symmetrische Verschlüsselung in CogniCrypt

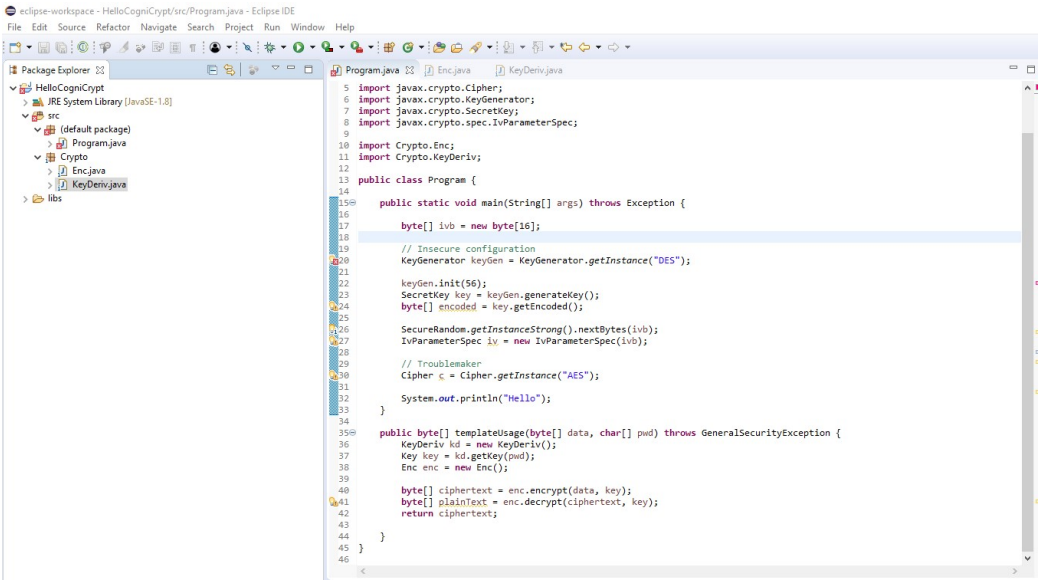

Abbildung 3.4: Projektstruktur nach der Codegenerierung durch CogniCrypt

mit Fehlern bewertet wird (s. Abbildung 3.5). Weitere Verbesserungsmöglichkeiten basieren auf den Nutzungserfahrungen des Autors und lassen sich in drei Kategorien einordnen.

IDE Integration

Die Installation von CogniCrypt erfolgt über den Menüpunkt „Install New Software...“ in Eclipse. Durch Eingabe der Projektadresse im Installationsassistenten wird das SCAT heruntergeladen und installiert. Aktuell wird CogniCrypt nicht über den Eclipse Marketplace, dem Erweiterungspaketmanager für Eclipse, vertrieben. CogniCrypt ist daher sehr implizit mit der IDE verbunden. Ob die Installation erfolgreich war und welche Version derzeit verwendet wird, kann nur unter dem Menüpunkt „About Eclipse IDE“ eingesehen werden. Dieser Punkt lässt jedoch gerade für unerfahrene Benutzer, wie auch dem Autor dieser Arbeit, nicht vermuteten dass hier Informationen für Plugins zu finden sind. Ein eigenes CogniCrypt-Menü ist nicht vorhanden. Des Weiteren stellt das Plugin keine Einstellungsmöglichkeiten bereit. Fehler im Code, die durch CogniCrypt identifiziert wurden, werden zudem nicht entsprechend ausgewiesen. Sind mehrere SCATs installiert, wäre beispielsweise eine Information für den Programmierer hilfreich, ob die betreffende Analyse durch CogniCrypt oder einem anderen Plugin erstellt wurde.

Erweiterbarkeit

Mit der Regelsprache CrySL wurde die Basis für eine Erweiterbarkeit von CogniCrypt geschaffen, die es zumindest von der Idee her erlaubt, zur Laufzeit der IDE neue Regeln mittels eines eigenen Texteditors zu schreiben und diese direkt auf die laufenden Codeanalysen anzuwenden. Derzeit werden diese Möglichkeiten jedoch nicht unterstützt. Um neue Analysen zu erhalten, ist man auf ein Update von CogniCrypt bzw. CrySL angewiesen. Weitere Mechanismen, die es erlauben würden CogniCrypt zu erweitern, sind nicht vorhanden. Ebenfalls ist es nicht möglich, einzelne Analysen durch CogniCrypt gezielt zu verwalten. Der Nutzer hat demnach keine Möglichkeit, einzelne Analysen, wie das Überprüfen der Instanzierung der `Cipher`-Klasse, abzuschalten. Eclipse bietet jedoch die Möglichkeit, Codeanalysen für Softwareprojekte per Konfigurationsdatei zu unterdrücken.

Will man nun eigene Erweiterungen für CogniCrypt entwickeln, ist man zunächst darauf angewiesen, die neue Regelsprache CrySL zu lernen. Das Veröffentlichen neuer Erweiterungen ist aktuell nur durch Erstellen eines Pull Requests auf der GitHub Seite des SCAT oder durch das Forken von CogniCrypt möglich.

Informationsgehalt

Durch Assistenten und sehr präzisen Codeanalysen versucht CogniCrypt den Programmierer bei der Verwendung der JCA zu unterstützen. Häufig wird dabei genau beschrieben, was CogniCrypt zu Lösung des Problems vorschlägt. Im Falle der Codegenerierung kann sich der Entwickler den Code zunächst einmal anzeigen lassen, bevor dieser in das Projekt integriert wird. Auch die Fehlermeldungen zeigen deutlich, an welchen Stellen der Code Fehler enthält und wie er zu beheben wäre. Wenig Informationen erhält der Programmierer jedoch, warum sein geschriebener Code nicht

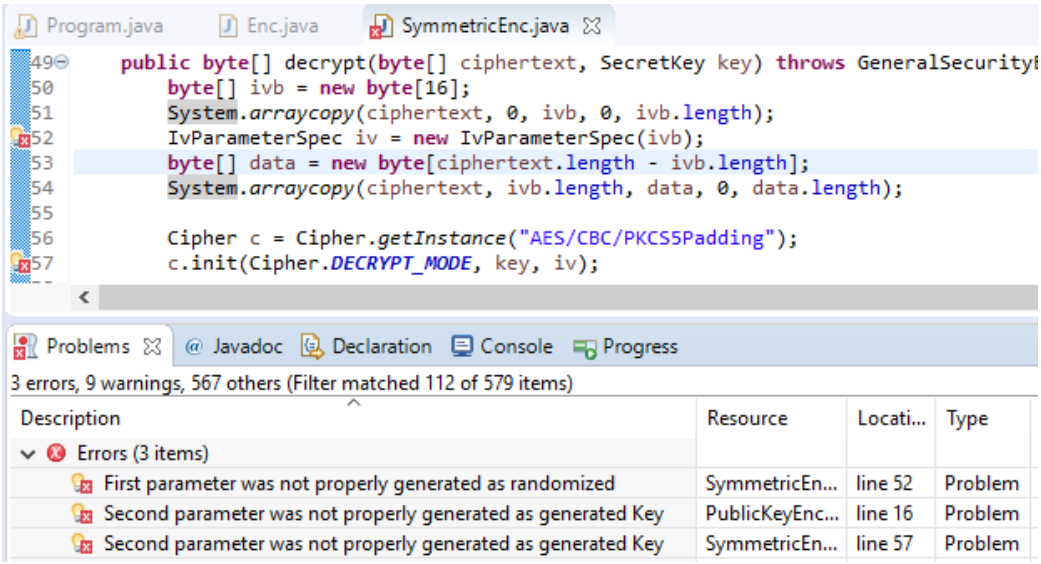

Abbildung 3.5: Fehlermeldung von generiertem Code durch CogniCrypt

sicher ist. Im oben aufgeführten Beispiel der Cipher Konfiguration wäre neben der Warnmeldung auch eine Information hilfreich, warum der Parameter "`AES`" nicht empfehlenswert ist. Ein Verweis auf die Standardkonfiguration mit dem unsicheren Modus Electronic Code Book (ECB), würde einem unerfahrenen Entwickler schon helfen, seinen Wissensstand zu erweitern.

3.6 Forschungsfragen

CogniCrypt stellt gewissermaßen ein Vorreiter in Sachen statischer Codeanalyse unter der Betrachtung der Nutzung von Krypto-APIs dar. Programmierfehler werden durch entsprechende Meldungen gekennzeichnet und erklärt. Durch zusätzliche Funktionen, wie das automatische Generieren sicherer Codefragmente, können Entwickler einfacher Programme schreiben. Was Krüger et al., 2017 nicht betrachtet haben, ist die Berücksichtigung anderer Programmiersprachen und deren zugehörigen Technologien. Mögen sich Java und C# in ihrer Syntax doch stark ähneln, so unterscheiden sich doch ihre APIs in Architektur, Konzepten und Schnittstellen deutlich. Demnach bleibt die Frage offen, ob und in welchem Ausmaß Ideen und Lösungen von CogniCrypt auf das C#-Ökosystem übernommen werden können.

Forschungsfrage 1 (FF1): In welchen Punkten unterscheidet und gleicht sich die JCA von der .NET Krypto-API?

Forschungsfrage 2 (FF2): Können die in CogniCrypt erarbeiteten Codeanalysen und Codegenerierungen auch für C# verwendet werden?

Aktuell bewertet CogniCrypt gefundene fehlerhafte Benutzungen der JCA als schweren Fehler. Die Ausgabe im Fehlerfenster der IDE Eclipse ist durch ein rotes Symbol entsprechend gekennzeichnet. IDEs besitzen jedoch die Möglichkeit, Auffälligkeiten im Code durch differenziertere Bewertungen anders darzustellen. So können beispielsweise bei leichten Fehlern nur Warnungen oder Empfehlungen ausgegeben werden. Bei

einer Evaluation aller Codemeldungen kann dann nach diesen Einstufungen gefiltert werden.

Forschungsfrage 3 (FF3): Können durch Konfiguration oder anderen Mechanismen Codemeldungen mit unterschiedlicher Einstufung durchgeführt werden?

Forschungsfrage 4 (FF4): Kann der Benutzer der IDE das Verhalten des SCAT ändern, um Analysen anders einzustufen, um somit auch eine bessere False Positive Rate oder False Negative Rate zu erlangen?

Für ein SCAT sind besonders Funktionen zur Erweiterbarkeit wichtig. Sie helfen ihren Anwendern neue Analysen und Funktionen in das Programm zu integrieren oder diese selbst zu entwickeln. *CogniCrypt* nutzt dafür die Regelsprache CrySL. Die C# Technologien, namentlich der Roslyn Kompiler, bieten ebenfalls Werkzeuge in Form eines eigenen Software Development Kit (SDK) an, die die Erweiterbarkeit eines SCAT ermöglichen. Das SDK kann jedoch nur über die .NET Programmiersprachen (C#, F# oder VB.net) angesprochen werden. Komponenten müssen also vor ihrer Verwendung kompiliert werden. Es stellt sich also die Frage, ob die Möglichkeit besteht, selbst erstelle Codeanalysekomponenten dynamisch, also zur Laufzeit des Analysetools bzw. der IDE, die die Analysen ausführt, einzufügen oder ob auch hier auf ein abstraktes Erweiterungsmodell wie CrySL zurückgegriffen werden muss.

Forschungsfrage 5 (FF5): Welche Erweiterungs- und Konfigurationsmöglichkeiten sollte ein Roslyn-basiertes SCAT der Nutzung von Krypto-APIs für Visual Studio besitzen?

Forschungsfrage 6 (FF6): Ermöglicht die IDE Visual Studio das dynamische Laden bzw. Einfügen weiterer Analysekomponenten?

3.7 Vorgehen bei der Literaturrecherche

Dieser Abschnitt beschreibt die konkrete Vorgehensweise bei der Literaturrecherche. Grundsätzlich lässt sich dieser Prozess in drei Phasen einteilen:

1. Sammeln von Literatur
2. Detaillierte Recherche und Kategorisierung der Literaturquellen
3. Recherche nach Bedarf

Die erste Phase begann mit der Auswahl des Themas dieser Arbeit. Hier wurde sich zunächst ein allgemeiner Überblick über das Thema verschafft, indem abstrakte Schlagwörter wie „statische Codeanalyse“, „Kryptographie“, „APIs“ oder „Visual Studio Erweiterungsentwicklung“ mittels der Internetsuchmaschine Google gesucht wurden. Diese Suche wurde sowohl auf Deutsch, als auch auf Englisch durchgeführt. Das Durchlesen der Resultate dieser Suche verschaffte einerseits ein allgemeines Verständnis der gesuchten Begriffe und brachte andererseits einen genaueren Überblick über verwandte Themen und bereits veröffentlichten Forschungen und wissenschaftlichen Arbeiten. Nachdem sich ein allgemeines Grundverständnis angeeignet

wurde und weitere relevante Schlagwörter herausgestellt wurden, begann das Sammeln der Literatur. Dazu wurden die Datenbanken Google Scholar, Researchgate, Springer Link, IEEE Xplore Digital Library und ACM Digital Library nach den bereits verwendeten, aber auch den neu erschlossenen Schlagwörtern befragt. Diese und auch alle weiteren Suchen wurden bis auf wenige Ausnahmen ausschließlich auf Englisch durchgeführt. Einige der neuen Suchbegriffe lauteten „secure software development“, „static code analysis false positive“ oder „usability crypto apis“. Wahlweise wurde die Schreibweise dieser Begriffe variiert, um neue Treffer zu erzwingen. Die aufgeführten Veröffentlichungen dieser Suche wurden zunächst nur gesammelt. Titel, Zusammenfassung und Jahr der Veröffentlichung waren die maßgeblichen Indikatoren, ob sie der Sammlung hinzugefügt wurde. Beim Veröffentlichungsdatum wurde darauf geachtet, hauptsächlich Literatur zu berücksichtigen, die innerhalb der letzten 15 Jahre veröffentlicht wurde. Ausnahmen wurden bei Literaturquellen gemacht, die grundlegende Sachverhalte beschreiben oder einen Grundstein für weitere Forschungen legten. Anhand der Literaturangaben der bereits gefundenen Arbeiten, konnten wiederum neue erschlossen werden. Es handelt sich demnach um eine Kombination der sogenannten Methodiken Schneeballprinzip und systematische Literaturrecherche[8]. Mit diesem Prinzip konnten Veröffentlichungen zu den zwei großen Themengebieten dieser Arbeit (Benutzung von Krypto-APIs und statische Codeanalyseprogramme) erschlossen werden. Die folgende Liste zeigt einige der wichtigsten Quellen zu diesen Themen:

> Benutzung von Krypto-APIs: (Krüger et al., 2018; Acar et al., 2017; Chatzikonstantinou et al., 2016; Nadi et al., 2016; Lazar et al., 2014; Egele et al., 2013; Robillard & DeLine, 2011; Stylos, Faulring, Yang & Myers, 2009)
>
> Statische Codeanalyseprogramme: (Krüger et al., 2017; Chess & West, 2007; B. Johnson et al., 2013; Baca et al., 2009; Ayewah et al., 2008; WASC, 2013; Bessey et al., 2010)

In der zweiten Phase wurde die bereits gefundene Literatur ausführlich durchgelesen. Wichtig erachtete Ausschnitte wurden in einer Textdatei unter dem Namen der Veröffentlichung zusammengetragen. Hierzu wurden die Originaltexte unübersetzt herauskopiert. Sofern das Lesen der Literatur weitere Quellen und Belege erforderte, wurden diese auf den bereits genannten Datenbanken gesucht und direkt gelesen. Das Resultat dieser Arbeit war eine unsortierte Sammlung potenziell wichtiger Aussagen. Um das spätere Verwenden dieser Aussagen zu erleichtern, wurden alle Textausschnitte in einer zweiten Textdatei nach Themen sortiert. Die nachstehende Auflistung zeigt diese Kategorisierung:

- Softwaresicherheit und Krypto-APIs
- statische Codeanalyseprogramme
- Softwareentwicklung und Softwaretests
- .NET, C#, Visual Studio und Roslyn

[8] https://www.bachelorprint.de/literaturrecherche/, Zugriff: 28.12.2018

Nach Abschluss dieser Phase wurden insgesamt fast 80 Literaturquellen bearbeitet.

Die dritte Phase der Literaturrecherche wurde ausschließlich beim Verfassen der Arbeit durchgeführt. Wurde für eine bestimmte Aussage oder Behauptung ein Beleg benötigt, wurden die Datenbanken, aber auch die Suchmaschine Google nach genau dieser Behauptung befragt. Verwies die Suche auf eine wissenschaftliche Arbeit, wurde sie als Quelle hinzugefügt. Verwies die Suche auf einen Artikel, wie beispielsweise Wikipedia, wurde versucht den wissenschaftlichen Ursprung, der in der Quelle gemachten Behauptung, zu finden. Konnte dieser nicht gefunden werden, wurden weitere Suchtreffer herangezogen. Neben wissenschaftlichen Veröffentlichungen, wurde zusätzlich, insbesondere für die Entwicklung von Sharper Crypto-API Analysis, auf die Dokumentationen des Visual Studio und Roslyn SDK zurückgegriffen.

Kapitel 4

Entwicklung von Sharper Crypto-API Analysis

Die in Abschnitt 3.4 herausgestellten Forschungsergebnisse lassen den Schluss zu, dass es in den Softwareentwicklungsprozessen erhebliches Verbesserungspotenzial gibt, die Programmierer durch geeignete Mittel zu unterstützen, Fehler beim Umgang mit Krypto-APIs zu vermeiden. Als mögliche Ursachen für unsichere Software wurden einerseits eine mangelnde Kompetenz mit dem Umgang von ihnen unbekannten Krypto-APIs (Nadi et al., 2016; Robillard & DeLine, 2011) sowie eine fehlerhafte Verwendung von Softwaretestmethoden festgestellt (Daka & Fraser, 2014; Uwano et al., 2006). Für jede dieser Ursachen auf sich allein gestellt, existieren bereits einige Verbesserungsansätze. Insbesondere für die fehleranfällige Benutzung von APIs gibt es verschiedenste Lösungsvorschläge (Clarke, 2004; Stylos et al., 2009). Für die Kombination dieser Ursachen sind dagegen bisher nur wenige Vorschläge unterbreitet worden. Durchaus als Vorbild kann die Arbeit von Krüger et al., 2017 angesehen werden, in der das SCAT CogniCrypt entwickelt wurde. Auch wenn CogniCrypt in Zukunft weitere Programmiersprachen und APIs unterstützen soll, liegt der Fokus momentan lediglich auf der JCA in Verbund mit der IDE Eclipse. Das in dieser Arbeit entwickelte SCAT, im weiteren Verlauf *Sharper Crypto-API Analysis* genannt, soll sich dagegen auf die Analyse von C#-Code spezialisieren. Mit Visual Studio wird eine Trägerplattform gewählt, die, wie Abschnitt 4.2 zeigen wird, im Vergleich zu Eclipse einen höheren Marktanteil besitzt und dadurch das Potential hat, noch mehr Entwickler anzusprechen. Ein besonderer Fokus der Entwicklung von Sharper Crypto-API Analysis wird auf die externe Erweiterbarkeit und individuelle Konfigurierbarkeit gelegt. Die Entwicklung einer Architektur, die diese Funktionalitäten bereitstellt, wird Teil dieser Arbeit sein. Aufgrund der geringen Betrachtung von C# und dem .Net-Ökosystem gibt der Abschnitt 4.2 eine detaillierte Beschreibung über die hier verwendeten Technologien.

Dieses Kapitel dokumentiert den Entwicklungsprozess von Sharper Crypto-API Analysis. Abschnitt 4.3 gibt eine detaillierte Auskunft über die funktionalen und nicht funktionalen Anforderungen für Sharper Crypto-API Analysis. Abschnitt 4.4 beschreibt sowohl die .NET Krypto-API und die in Sharper Crypto-API Analysis verfügbaren Codeanalysen. Abschnitt 4.5 dokumentiert den Entwicklungsprozess und geht dabei auf einige technische Details ein, die zum besseren Verständnis über Architektur und Funktionsweise von Sharper Crypto-API Analysis beitragen sollen.

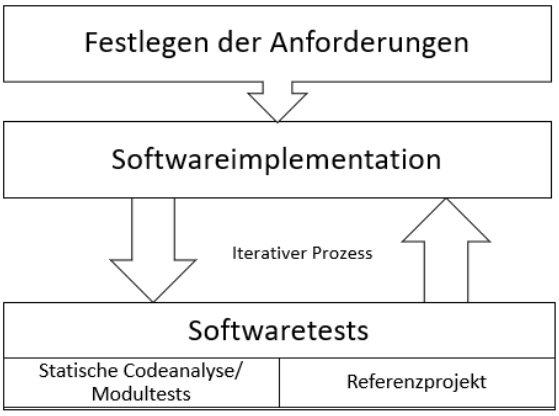

Abbildung 4.1: Software Development Life Cycle für Sharper Crypto-API Analysis (Eigene Darstellung)

4.1 Vorgehensweise

Bevor mit dem Programmieren des Tools begonnen werden kann, müssen zunächst weitere, für die Softwareentwicklung wichtige Aufgaben durchgeführt werden. Dieser Abschnitt beschreibt kurz die allgemeine Vorgehensweise, die für die Entwicklung von Sharper Crypto-API Analysis angewendet wurde. Die einzelnen Entwicklungsabschnitte werden folglich in den Abschnitten 4.3 bis 4.5 dokumentiert.

Die Entwicklung von Sharper Crypto-API Analysis richtet sich im wesentlichen nach den Aktivitäten, die von der ISO Norm 12007 unter dem Punkt „Software Implementation Processes“ („ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes“, 2008) vorgegeben werden. Besonderer Fokus in dieser Arbeit liegt jedoch auf der Aktivität des Softwareherstellungsprozesses. Die Abbildung 4.1 zeigt den SDLC, der für diese Arbeit angewendet wurde. Als erste Aktivität müssen die Anforderungen festgelegt werden, die in Sharper Crypto-API Analysis implementiert werden sollen. Entscheidende Vorarbeit liefern die in Kapitel 3 erarbeiteten Forschungsergebnisse. Sie stellen die Grundlage für sämtliche Entscheidungen und Priorisierungen der Anforderungsanalyse (Software Requirements Analysis Process) bereit. In Abschnitt 4.3 werden sie in funktionale und nichtfunktionale Anforderungen dokumentiert und entsprechend ihrer Wichtigkeit priorisiert. Eine Priorisierung wird besonders für Projekte mit knappem Zeitplan empfohlen, da einerseits die wichtigsten Funktionalitäten herausgestellt werden, andererseits gibt sie auch eine Entscheidungshilfe, sollten zwei oder mehrere Anforderungen konfliktäre Eigenschaften haben (Paetsch, Eberlein & Maurer, 2003, S. 2).

Im nächsten Schritt wird das statische Codeanalyseprogramm Sharper Crypto-API Analysis entwickelt (Software Construction Process). Eine Beschreibung dieses Prozesses erfolgt in Abschnitt 4.5. Die Analysen beziehen sich auf die API-Spezifikation des .NET-Standard 2.0. Die Krypto-API ist unter dem Namespace `System.Security.Cryptography` verfügbar. Das SCAT wird auf der Compilerplattform Roslyn aufbauen, mit welcher auch Visual Studio selbst ausgerüstet ist. Ferner wird Sharper Crypto-API Analysis als Plugin für die IDE Visual Studio 2017 unter Windows verfügbar sein. Eine nähere Beschreibung und Verwendungsbegründung der verwendeten Technologien erfolgt in Abschnitt 4.2. Parallel zur Entwicklung von Sharper

Crypto-API Analysis werden regelmäßig Softwaretestprozesse (Software Qualification Testing Process) durchgeführt. Anhand dieser Tests wird einerseits die Funktionalität des Tools sichergestellt. Gefundene Fehler werden nach Bedarf direkt korrigiert. Andererseits soll zum Abschluss der Arbeit, mit Hilfe eines Referenzprojekt, eine kurze Einschätzung der Qualität des Tools und Ziele der Arbeit erfolgen. Abschnitt 5.4 widmet sich dieser Aufgabe.

4.2 Verwendete Technologien

Programmiersprache C#

C# ist eine von Microsoft entwickelte, objektorientierte, Programmiersprache. Ihre aktuelle Version ist 7.3. Die Versionen 1.0, 2.0 und 5.0 sind von der Normungsorganisation EMCA spezifiziert[1]. Anhand von Indizes lässt sich die Popularität einer Programmiersprache ablesen. Einer der bekanntesten Anbieter ist der TIOBE-Index[2]. Stand Dezember 2018 liegen Java und C weit vorne. Darauf folgen C++ und Python. Auf dem sechsten Platzt liegt die Programmiersprache C#. Beim PYPL-Index[3] liegt C# sogar auf dem vierten Platz. Nicht nur bei den Programmierern, sondern auch in vielen wissenschaftlichen Untersuchungen, spielen besonders Java und C/C++ eine entscheidende Rolle. Andere Sprachen werden selten erwähnt und noch seltener explizit behandelt. Allein die bisher hier aufgeführte Literatur über SCATs, beschäftigt sich ausschließlich mit Java oder C. Eine ausführlichere Betrachtung dieser sehr populären Programmiersprache, einschließlicher ihrer zugehörigen Technologien, ist daher aus Forschungsgründen durchaus relevant.

.NET Produktfamilie

Programme der Sprache C# werden hauptsächlich in einen Zwischencode namens Common Intermediate Language (CIL) übersetzt. Dieser Zwischencode wird dann durch eine Laufzeitumgebung Just-in-time (JIT) kompiliert und ausgeführt. Sowohl CIL und Laufzeitumgebung sind in einem ISO[4] und ECMA[5] Standard spezifiziert. Für die CLI gibt es mehrere Implementationen, darunter die unten aufgeführten:

- **.NET Framework**. Das .NET Framework ist die erste öffentliche Implementierung der CLI. Seine Laufzeitumgebung wird Common Language Runtime (CLR) genannt. Neben dieser stellt das .NET Framework auch die Standard-API und weitere Funktionskomponenten verfügbar. Zu diesen gehören unter anderem die GUI-Implementierungen WinForms und die Windows Presentation Foundation (WPF). Die Standard-API implementiert zentrale Komponenten, wie die Klasse `Object` oder die IO-Schnittstelle zum Betriebssystem. Das .NET Framework ist nur für Windows verfügbar.

[1] https://www.ecma-international.org/publications/standards/Ecma-334.htm, Zugriff: 30.12.2018
[2] https://www.tiobe.com/tiobe-index/, Zugriff: 09.12.2018
[3] http://pypl.github.io/PYPL.html, Zugriff: 09.12.2018
[4] https://www.iso.org/standard/58046.html, Zugriff: 30.12.2018
[5] https://www.ecma-international.org/publications/standards/Ecma-335.htm, Zugriff: 30.12.2018

- **.NET Core**. .NET Core ist die neuste Implementierung der CLI. Sie ist quelloffen und plattformunabhängig. Die Laufzeitumgebung wurde im Vergleich zum .NET Framework angepasst und bekam unter anderem einen neuen JIT-Kompiler. Die Standard-API wurde teilweise dem .NET Framework entnommen und gegebenenfalls an einigen Stellen, wie zum Beispiel dem IO-Handling angepasst, um die verschiedenen Betriebssysteme zu unterstützten.

Um zwischen den einzelnen Implementierungen Kompatibilität zu schaffen, wurde .NET Standard eingeführt. „.NET Standard ist eine formale Spezifikation von .NET-APIs, die für alle .NET-Implementierungen verfügbar sein sollen.“[6]. Das bedeutet, dass alle Implementierungen einer CLI, die den Standard erfüllen wollen, eine API anbieten müssen, welche alle Schnittstellen des .NET Standard implementiert. Die aktuelle Version ist 2.0 und beinhaltet zum Beispiel auch Schnittellen für kryptographische Methoden. Zudem breitet sich die .NET-Familie nicht nur auf den Desktopplattformen Windows, MacOS und Linux aus, sondern erhält dank der *Xamarin Platform* (Sammlung von Softwarebibliotheken und Kompilern) auch Einzug auf die mobilen Systeme Android und iOS. Damit sind eigentlich alle Plattformen abgedeckt, die vom Endverbraucher benutzt werden.

Roslyn Compiler

Eine weitere wichtige Technologie der .NET-Produktfamilie ist *Roslyn*. Roslyn ist der quelloffene Kompiler aller .NET kompatiblen Programmiersprachen (u. a. C#, F#, Visual Basic .NET) Programmiersprachen[7]. Er übersetzt den Quellcode dieser Sprachen in die CIL. Neben dem eigentlichen Kompilieren bietet Roslyn APIs an, mit denen Entwickler Codeanalysen und Quick-Fixes schreiben können. Diese, wie auch der Kompiler selbst, können als Erweiterung für die IDE Visual Studio installiert werden. Eine einzelne Roslyn-Komponente (Codeanalyse oder Quick-Fix) wird auch als *Analyzer* bezeichnet.

Entwicklungsprogramme

Die Implementation von Sharper Crypto-API Analysis erfolgt mit der IDE Visual Studio und dessen eigenem SDK zur Erstellung von Erweiterungen. Laut einer Umfrage auf der Webseite Stackoverflow.com ist Visual Studio mit einem Anteil von 34,3% nach Visual Studio Code (34,9%), die am zweithäufigsten benutze IDE. Die erste für Java spezialisierte IDE, IntelliJ, liegt mit 24,9% dagegen nur auf dem sechsten Platz[8]. Ein auf der Googlesuche basierter Index bewertet Visual Studio mit einem Anteil von 23,73% auf Platz eins[9] Es bietet sich daher an, Sharper Crypto-API Analysis als Plugin für Visual Studio und nicht als eigenständiges Programm bereitzustellen. Aus den in Abschnitt 3.3.2 benannten Forschungen geht hervor, dass eine hohe Nutzererfahrung eines SCAT zu einer höheren Produktivität führt. Ein SCAT, das direkt in eine IDE integriert ist, hat daher das große Potential, dass der Benutzer mit seinen Interaktionspunkten, wie der Fehlerausgabe und Bedienelementen, schnell vertraut

[6] https://docs.microsoft.com/de-de/dotnet/standard/net-standard, Zugriff: 30.12.2018
[7] https://github.com/dotnet/roslyn, Zugriff: 30.12.2018
[8] https://insights.stackoverflow.com/survey/2018/#technology-most-popular-development-environments, Zugriff: 30.12.2018
[9] https://pypl.github.io/IDE.html, Zugriff: 30.12.2018

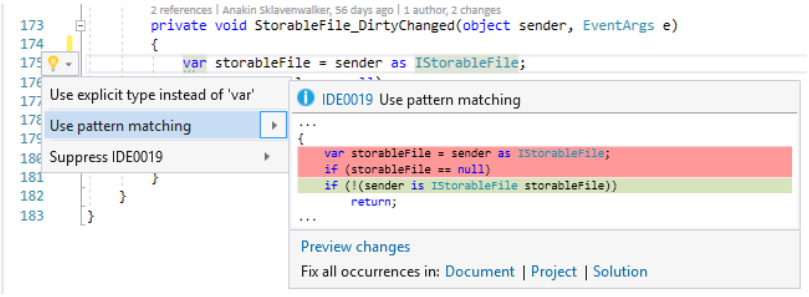

Abbildung 4.2: Benutzeroberfläche für Codeverbesserungen in Visual Studio

wird (Chess & West, 2007, S. 136). Plugins für Visual Studio haben den Vorteil, dass diese auf sämtliche UI-Komponenten der IDE zugreifen können. Im Falle von Sharper Crypto-API Analysis hat dies beispielsweise zur Folge, dass die UI zur Identifikation von Fehlern durch unterkringelten Text und die UI für Codeverbesserungen genutzt werden können. Die Abbildung 4.2 zeigt diese Interaktionskomponenten. Das Glühbirnensymbol signalisiert, dass ein oder mehrere Codemeldungen in dieser Zeile verfügbar sind. Zusätzlich wird das entsprechende Token (ein lexikalisches Element der Programmiersprache) im Code gekennzeichnet, indem es grau unterpunktet wird. Je nach Konfiguration kann das Token auch in anderen Farben unterkringelt werden. In diesem Beispiel sind zwei Verbesserungen verfügbar. Bevor der Entwickler diesen Vorschlag anwendet, bekommt er eine Vorschau angezeigt, die darstellt, wie und welche Codeteile geändert werden.

Als unterstützende Maßnahme zur Entwicklung wird Resharper als statisches Codeanalyse- und Refektoringtool benutzt. Dadurch kann eine Konsistenz des Codes sowie das Einhalten von Codekonventionen sichergestellt werden. Das Codemanagement wird durch das hochschulinterne Gitlab Repository dieser Masterarbeit verwaltet. Bereitgestellt wird das Plugin in Form einer Installationsdatei für Visual Studio Erweiterungen auf dem Repository und der Begleit-CD dieser Arbeit.

4.3 Anforderungen

Dieser Abschnitt befasst sich mit der Formulierung und Begründung der Anforderungen von Sharper Crypto-API Analysis. Dazu wird zunächst die Zielgruppe für dieses Tool festgelegt. Anschließend werden die Rahmenbedingungen dieser Anforderungsanalyse aufgeführt. Abschließend erfolgt eine Auflistung und Begründung der wichtigsten Anforderungen.

4.3.1 Bestimmung der Zielgruppe

Die statische Codeanalyse kann sowohl während der Implementierungsphase, als auch in der Testphase der Softwareentwicklung eingesetzt werden. Je nach Phase und Unternehmensstruktur, kann sich auch der Anwender eines SCAT unterscheiden. Sind es bei der Implementierungsphase noch Programmierer, die ein SCAT nutzen, können in der Testphase auch vermehrt Sicherheitsexperten und Qualitätsprüfer Benutzer des SCAT sein. Die Anforderungen an das Tool unterscheiden sich dementsprechend. Aus den in Kapitel 3 ermittelten Ergebnissen geht beispielsweise hervor, dass Programmierer mit einer hohen False Positive Rate überfordert werden. Sicherheitsexperten

hingegen sind in der Lage, diese Fehlalarme von richtig erkannten Analysen zu unterscheiden. Aufgrund des begrenzten Zeitrahmens dieser Arbeit, soll sich zunächst auf eine Anwendergruppe, der Zielgruppe von Sharper Crypto-API Analysis, konzentriert werden. Die in der Einleitung genannten Beispiele zeigen nicht nur, wie scheinbar einfach Schwachstellen durch eine falsche Nutzung der Krypto-APIs entstehen können, sondern auch, dass diese mit wenig Aufwand behoben werden können.

Zur Veranschaulichung wird erneut das Beispiel der Verwendung des DES Verschlüsselungsalgorithmus' betrachtet. In der .NET Krypto-API erben sowohl die Klasse `DES`, als auch `AES` von der abstrakten Klasse `SymmetricAlgorithm`. Das bedeutet, dass sämtliche Deklarationen von Konstruktoren, Methoden und Eigenschaften in beiden Implementationen identisch sind. Ein Wechsel von der veralteten DES Verschlüsselung zu seinem Nachfolger AES kann also einfach erfolgen, indem lediglich die Zuweisung der Variable ausgetauscht wird. Listing 4.1 verdeutlicht diese Aufgabe. Zum Austausch des Algorithmus' muss lediglich die zweite Zeile mit der achten Zeile – ohne Kommentierung – ersetzt werden[10].

```
1 // DES based encryption
2 using (var cryptoAlgorithm = new DESCryptoServiceProvider())
3 {
4     // perform encryption here...
5 }
6
7 // change for AES based encryption
8 // using (var cryptoAlgorithm = new AesManaged())
```

Listing 4.1: Austausch der Verschlüsselungsalgorithmen DES und AES in C#

Je früher der Fehler gefunden und behoben werden kann, desto weniger Zeit, Aufwand und Geld kostet diese Aufgabe. Die Programmierer dieses Codes sind daher auch in der besten Position den Fehler zu korrigieren. Die Zielgruppe von Sharper Crypto-API Analysis wird daher für diese Arbeit primär auf die Programmierer in der Implementationsphase und weniger auf Entwickler in der Testphase festgelegt.

4.3.2 Formulierung der Anforderungen

Nachdem die Zielgruppe bestimmt wurde, können nun die einzelnen Anforderungen an Sharper Crypto-API Analysis formuliert werden. Zunächst wird allgemein in funktionale und nicht-funktionale Anforderungen unterschieden. Die funktionalen Anforderungen beschreiben dabei das Funktionsangebot des Tools unter Berücksichtigung der Ein- und Ausgaben sowie das Verhalten des Programms (Partsch, 2010, S. 27). Die nicht-funktionalen Anforderungen beschreiben die Qualitätseigenschaften des SCAT (Grande, 2014, S. 39). Aufgrund des Fokus' dieser Arbeit, sich auf die Entwicklung des SCAT zu konzentrieren und dem begrenzen Zeitumfang, wird hier von einem vollständigem Anforderungsentwicklungsprozess abgesehen. Ebenfalls ist es nicht Anspruch dieser Arbeit, einen voll umfänglichen Anforderungskatalog anzufertigen. Die in diesem Abschnitt formulierten Anforderungen ergeben sich aus der Literaturrecherche dieser Arbeit. Die Abschließende Priorisierung der Anforderungen basiert

[10]Der Vollständig halber soll angemerkt werden, dass der Austausch des Verschlüsselungsalgorithmus' natürlich nicht automatisch sämtliche Schwachstellen behebt.

einerseits der auf der Quantität, wie oft eine Anforderung in verschiedenen Quellen gefunden wurde und andererseits auf der Umsetzbarkeit im zeitlichen Rahmen dieser Arbeit.

Die Tabelle im Anhang A zeigt alle für Sharper Crypto-API Analysis formulierten Anforderungen. Zu den Angaben gehören einerseits eine Funktionsnummer, mit welcher diese Anforderung im weiteren Verlauf der Arbeit bezeichnet wird. Die Spalte „Bezeichnung“ gibt den Namen der Anforderung an und die Spalte „Beschreibung“ gibt benötigte Informationen, wie die Verhaltensweise der Funktionalität an. Ob die Anforderung zum Zeitpunk der Abgabe dieser Arbeit implementiert ist, gibt die Spalte „Status“ an. Weitere Informationen sind in „Bemerkungen“ aufgeführt.

Zur besseren Übersicht wurden alle Anforderungen in große Kategorien eingeteilt:

- **Codeanalyse:** Diese Kategorie befasst sich mit allen allgemeinen funktionalen Anforderungen der Codeanalysefunktion des SCAT. Hier werden jedoch nicht die einzelnen Codeanalysen aufgeführt, die Sharper Crypto-API erfüllen soll. Diese Spezifikation erfolgt im Abschnitt 4.4.
- **Quick-Fixes:** Diese Kategorie befasst mit den funktionalen Anforderungen, die das Bereitstellen von Quick-Fixes betreffen.
- **Codegenerierung:** Diese Kategorie befasst sich mit allen funktionalen Anforderungen, welche die Codegenerierung in Shraper Crypto-API Analysis betreffen.
- **Konfiguration:** Funktionen, mit denen man das SCAT konfiguriert, sind in dieser Kategorie zu finden.
- **Erweiterbarkeit durch den Benutzer:** Diese Kategorie befasst sich mit den funktionalen Anforderungen zur Erweiterbarkeit des SCAT durch den Anwender.
- **Erweiterbarkeit durch Erweiterungsentwickler:** Diese Kategorie befasst sich mit den funktionalen Anforderungen zur Erweiterbarkeit des SCAT durch einen Erweiterungsentwickler.
- **IDE Integration:** Diese Kategorie beschreibt Qualitätseigenschaften von Sharper Crypto-API Analysis im Zusammenspiel mit der IDE.
- **Bedienbarkeit:** Qualitätseigenschaften, welche die Interaktion mit dem Anwender betreffen, sind hier gelistet.
- **Benutzeroberfläche:** Qualitätseigenschaften, welche speziell die Benutzeroberfläche betreffen, sind hier gelistet.

Die wichtigsten Anforderungen dieser Kategorien werden im weiteren Verlauf dieses Abschnitts detaillierter begründet.

Codeanalyse

Die Durchführung von Codeanalysen ist die Hauptfunktion eines jeden SCAT. Damit das Tool allerdings auch von der Zielgruppe akzeptiert wird, ist eine niedrige False Positive Rate bei der Klassifizierung der Codeanalyse von entscheidender Wichtigkeit (Bessey et al., 2010, S. 75). Um die Codeanalysen für die Arbeitsprozesse des Programmierers effizient durchzuführen, bedarf es einiger Anforderungen:

Die Unterstützung der Programmiersprache C# und die Fokussierung der .NET Krypto-API sind für Sharper Crypto-API Analysis zwingend erforderlich. Die Codeanalysen sind auf Basis des vom Anwender geschriebenen Quellcodes durchzuführen. Eine Codeanalyse der CIL hat den Nachteil, dass die Entwickler gefundene Analyseresultate vom Zwischencode auf ihren eigenen Quellcode übertragen müssen. Siehe Anforderung **F1.0**.

Des Weiteren bringt die Untersuchung des Quellcodes, anstelle des Bytecodes den Vorteil mit sich, dass die Analysen auch dann durchgeführt werden können, wenn sich der Code noch nicht kompilieren lässt (Baca et al., 2008, S. 81; Chess & West, 2007, S. 51). Siehe Anforderung **F1.1**.

Entwickler fühlen sich durch eine zu große Menge an Warnmeldungen, darunter auch False Positives, im Code gestört. Sie äußern daher den Wunsch, diese, wenn auch nur vorübergehend, gänzlich, oder nur in bestimmten Codebereichen ausblenden zu können (B. Johnson et al., 2013, S. 677). Um dem Anwender des SCAT die Möglichkeit zu geben False Positive Meldungen eigenständig zu verwalten oder für ihn nicht relevante Meldungen auszublenden, ist eine Funktion zu implementieren, die es dem Benutzer ermöglicht Codeanalysen zu unterdrücken (Chess & West, 2007, S. 108; WASC, 2013, S. 11). Siehe Anforderung **F1.2**.

Das erste Mittel, um Fehler bei der Verwendung von Krypto-APIs zu vermeiden, ist das Vermitteln von Wissen (Howard & Lipner, 2003, S. 58). Analysemeldungen im Code, müssen daher mit Informationen angereichert werden, die dem Entwickler nachhaltig helfen, das durch den fehlerhaften Code entstandene Problem zu verstehen und wie es zu beheben ist. Die Codeanalyse ist in Form eines Berichtes dem Nutzer mitzuteilen. Dieser muss mindestens die folgenden Informationen enthalten:

- **Analyse ID**: Ein eindeutiger Fehlercode.
- Eine **Beschreibung** des Fehlers mit technischen Details.
- **Warnstufe**: Die von Visual Studio unterstützten Warnstufen: Fehler, Warnung und Mitteilung
- **Analyzer**: Die Analysekomponente, die den Fehler gemeldet hat.
- **Dateiname** und **Zeilennummer**.

Ferner geht aus Befragungen von Programmierern hervor, dass sie zusätzliche Informationen wie Internetlinks oder Lösungsvorschläge hilfreich finden (B. Johnson et al., 2013, S. 677). Siehe Anforderungen **F1.3** und **F1.5**. Diese und hier neue vorgeschlagene Punkte sind daher ebenfalls in Analyseberichten einzufügen:

- Eine **Einordnung** des gefundenen Fehlers in Bezug auf Bedrohungen und Schweregrad.

- **Lösungsbemerkungen** oder **Lösungsvorschläge**, die den fehlerhaften Code betreffen.
- Zusätzliche informative **Links** zu Artikeln o.ä., die diesen Fehler, die Bedrohung oder Verwundbarkeit aufzeigen.

Eine zusätzliche Anforderung, die für diese Arbeit zunächst nicht zentral im Fokus steht, ist die Interoperabilität der Analyseberichte mit anderer Software, wie beispielsweise einem Webbrowser (Kleidermacher, 2008, S. 368). Siehe Anforderungen **F1.6** und **F1.7**.

Quick-Fixes

Entwickler zeigen Probleme auf, die Fehler zu beheben, die ihnen durch SCATs gemeldet wurden. So berichtete beispielsweise ein Programmierer, dass ein SCAT, wenn es schon in der Lage war den Fehler zu finden, diesen auch beheben können sollte (B. Johnson et al., 2013, S. 677). Quick-Fixes sind Funktionen mit denen SCATs gefundene Fehler durch Generierung kleiner Codefragmente eigenständig beheben können. Untersuchungen konnten belegen, dass diese Funktionalität die Gebrauchstauglichkeit eines SCAT steigert. Voraussetzung sind jedoch einerseits, ein möglichst übersichtliches Angebot an vorhandenen Quick-Fixes. Andererseits wollen Entwickler auch, bevor sie den Quick-Fix anwenden, eine Vorschau sehen, in wie weit sich ihr Code verändert (B. Johnson et al., 2013).

Visual Studio im Zusammenspiel mit Roslyn bietet all diese Funktionalitäten an. Sharper Crypto-API Analysis soll daher auf dieses Funktionsangebot zurückgreifen und in geeigneten Fällen selbst Quick-Fixes anbieten. Siehe Anforderungen **F2.0** bis **F2.2**.

Codegenerierung

Der Abschnitt 3.2 schildert, dass die mangelnde Projektion von kryptographischen Aufgaben, wie dem Verschlüsseln von Nachrichten auf die vorhandenen Funktionen der Krypto-API ein Grund ist, warum Programmierer Schwierigkeiten beim Verwenden dieser APIs haben. Das SCAT wirkt dem mit einer Funktion zur Codegenerierung von kryptographischen Aufgaben entgegen. In der Veröffentlichung zu CogniCrypt wird die Vorgehensweise der Codegenerierung beschrieben (Krüger et al., 2017, S. 931). Zentrales Element ist ein Assistent, der dem Programmierer bei der Auswahl und Konfiguration seiner Aufgabe hilft. Der Abschnitt 3.5.2 beschreibt diesen Prozess ausführlich. Sharper Crypto-API Analysis nimmt sich CogniCrypt daher zum Vorbild und implementiert diese Funktionalität entsprechend für die Generierung von C# Code. **F3.0** bis **F3.5** spezifizieren die Anforderungen für diese Funktionalität. Der Abschnitt 4.5.4 befasst sich genauer mit der technischen Umsetzung dieser Anforderungen.

Konfigurierung

Die Integration des SCAT in den Entwicklungsprozess hilft dem Entwickler, bessere Resultate bei der Arbeit zu erzielen. Zum Entwicklungsprozess gehört allerdings nicht nur das Schreiben des Quellcodes, sondern auch die Quellcodeverwaltung. Wird

Software im Team entwickelt, kommt häufig das Versionsverwaltungsystem Git zum Einsatz[11]. Neben dem Quellcode bietet es sich auch an Konfigurationseinstellungen der Entwicklungstools zu verwalten. Die Idee dahinter ist, dass sämtliche Entwicklungsrechner der Programmierer direkt und ohne viel Aufwand konfiguriert werden können.

Sharper Crypto-API Analysis soll eine Architektur zur Verfügung stellen, welche es erlaubt, Konfigurationen aus einem gemeinsam genutzten Git-Repository zu beziehen und entsprechend anzuwenden. Neben Einstellungsmöglichkeiten wie der Analyseerkennungsrate, sollen so auch installierte Analysekomponenten verwaltet werden können. Siehe Anforderungen **F4.2** und **F4.3**. Der Abschnitt 4.5.5 beschreibt weitere Details und Begründungen für diese Funktion.

Erweiterbarkeit durch den Benutzer

Neue Bedrohungen und Schwachstellen werden ständig gefunden. Ein SCAT muss daher in der Lage sein, seine Analysekomponenten zu aktualisieren (Bardas, 2010). Siehe Anforderung **F5.0**. Neben Aktualisierungen müssen noch weitere Verwaltungsfunktionen für Analysekomponenten verfügbar sein (siehe Anforderung **F5.2**):

- Eine **Ansicht** aller installierter und verfügbarer Analysekomponenten ist erforderlich.
- Analysekomponenten müssen dem Tool **hinzugefügt** werden können.
- Installierte Komponenten müssen sich **löschen** lassen.
- Installierte Komponenten müssen sich **updaten** lassen.

Um auch zukünftig Programmierfehler in Sharper Crypto-API Analysis zu beheben, ist es ebenfalls notwendig, dass die Visual Studio Erweiterung aktualisiert werden kann. Microsoft bietet dazu einen Marktplatz an, auf dem Erweiterungen für Visual Studio verfügbar sind. Updates der Plugins werden ebenfalls über diesen Marktplatz verwaltet. Siehe Anforderung **F5.1**.

Erweiterbarkeit durch den Erweiterungsentwickler

Dieser letzte große Punkt der funktionalen Anforderungen legt fest, Sharper Crypto-API Analysis in einer Architektur zu entwickeln, die es anderen Programmieren erlaubt, Erweiterungen für dieses SCAT zu entwickeln. Grundsätzlich sollen alle hier aufgeführten Funktionen durch ein SDK erweitert werden können. Darunter fallen auch:

- Schreiben neuer Codeanalysen
- Schreiben neuer Code-Fixes
- Hinzufügen neuer kryptographischer Aufgaben zur Codegenerierung
- Anpassen der Analyseberichte

[11] https://www.openhub.net/repositories/compare, Zugriff: 11.12.2018

- Hinzufügen neuer Einstellungsmöglichkeiten, um etwa die Erkennungsrate des SCAT zu ändern.

Damit das SDK effizient genutzt werden kann, muss dieses dokumentiert sein. Siehe Anforderungen **F6.0** und **F6.1**

IDE Integration

Die Integration eines SCAT in eine IDE hat für den Programmierer viele Vorteile. Sie sind mit ihrer IDE vertraut und wissen, wo bestimmte Funktionen oder Einstellungsmöglichkeiten zu erwarten sind. Die Installation eines Plugins sowie dessen Konfiguration, müssen sich an die Benutzerinteraktionskonzepte halten, die von der IDE vorgegeben werden, um hier Problemen vorzubeugen (B. Johnson et al., 2013, S. 676). Siehe Anforderung **NF1.0**.

Um den Arbeitsfluss des Entwicklers nicht zu stören, darf das SCAT die die Arbeitsabläufe nicht aufhalten, die IDE nicht blockieren oder zum Absturz bringen (Chelf & Ebert, 2009). Siehe Anforderungen **NF1.1** und **NF1.2**.

Ferner soll festgelegt werden, dass eine erfolgreiche Installation des SCAT in Visual Studio deutlich kenntlich zu machen ist. Somit wird der Schwierigkeit festzustellen, ob CogniCrypt erfolgreich in Eclipse integriert wurde (s. Abschnitt 3.5.3), entgegengewirkt. Siehe Anforderung **NF1.3**.

Bedienbarkeit und Benutzeroberfläche

Sharper Crypto-API Analysis ist ein für Programmierer ausgelegtes SCAT. Für einen effizienten Umgang ist es daher essentiell, dass die Interaktion mit dem SCAT auch auf diese Zielgruppe und nicht etwa auf Sicherheitsexperten ausgelegt ist (Chess & McGraw, 2004; B. Johnson et al., 2013). Um dieses Ziel zu erreichen, sind einige Anforderungen wichtig.

Sämtliche Elemente der Benutzeroberfläche müssen aus Windows oder Visual Studio entnommen oder an diese angelehnt sein. Programmierer haben dadurch den Vorteil, dass sie die einzelnen Bedienkonzepte des SCAT bereits kennen. Der Lernaufwand ist daher gering (Chess & West, 2007, S. 136). Siehe Anforderung **NF3.0**.

Analyseberichte und Meldungen müssen mit dem Vokabular eines C# Programmierer geschrieben sein. Bedrohungen und Schwachstellen müssen auf eine Weise erklärt werden, die es dem Programmierer ermöglichen, diese mit nur einem geringen Fachwissen zu verstehen (B. Johnson et al., 2013). Siehe Anforderung **NF3.1**.

4.4 Analyzers der Nutzung der .NET Krypto-API

Dieser Abschnitt beschreibt, welche Analyzer, im aktuellen Entwicklungsstand der Arbeit in Sharper Crypto-API Analysis umgesetzt wurden. Zur besseren Einordnung der Analyzer, werden zunächst die in Abschnitt 2.2.3 vorgeschlagenen Erkennungsklassen für SCATs erörtert. Anschließend erfolgt eine Abgrenzung des für Sharper Crypto-API Analysis entwickelten Funktionsumfang sowie eine kompakte Beschreibung des Funktionsangebot der .NET Krypto-API. Abschließend werden die hier implementierten Analyzer kurz vorgestellt.

4.4.1 Erkennungsklassen für SCATs der Nutzung von Krypto-APIs

Aufgrund der mangelnden Berücksichtigung von Krypto-APIs bei den Funktionsbeschreibungen von SCATs, werden in Abschnitt 2.2.3 fünf neue Erkennungsklassen vorgeschlagen, welche sich auf die Codeanalysen der Nutzung von Krypto-APIs spezialisieren. Zur besseren Übersicht werden sie hier noch einmal aufgeführt.

- Schwache Verschlüsselungsalgorithmen
- Schwache Hashingalgorithmen
- Schwache Konfiguration der benutzten API
- Verstoß gegen die API-Spezifikation
- Auslesbare Informationen

Der Ursprung dieser Erkennungsklassen geht größtenteils aus der für diese Arbeit untersuchten Literatur hervor. Andere Quellen sind Erfahrungen des Autors, Suchergebnissen auf Programmierseiten wie Stackoverflow.com nach schwachen Nutzungen der .NET Krypto-API und CVE- und Common Weakness Enumeration (CWE)-Einträgen.

Zu den Suchergebnissen auf den Programmierseiten zählen unter anderem die in der Einleitung aufgeführten Beispiele auf Codeproject.com. Weitere dieser Artikel wurden ebenfalls gefunden[12]. Auch in diesem Fall rät der Autor des Artikels, den geheimen Schlüssen im Quelltext per `string` Variable zu kodieren. Auch hier wird der IV mit dem Schlüssel gleichgesetzt.

Mittels CVEs werden veröffentlichte Schwachstellen und Verwundbarkeiten eingetragen. Die CWE listet hingegen häufig vorkommende Schwachstellen in Software auf und klassifiziert diese. Die relevante Kategorie für diese Arbeit ist die CWE-310[13], welche kryptographische Fehler behandelt. Unter anderem werden dort die Klassen „Unangebrachte Verschlüsselungsstärke“ oder „Verwendung von gebrochenen oder risikoreichen kryptographischen Algorithmen“ aufgeführt. Generell könnte man diese Taxonomie der CWE bereits als Erkennungsklassen für SCATs heranziehen. Eine präzisere Abstufung, wie in dieser Arbeit vorgeschlagen, erscheint jedoch sinnvoll, um einerseits genauer aufzeigen zu können, welche Bedrohungen schlecht oder falsch genutzte einzelne kryptographische Verfahren besitzen und andererseits, um den Benutzern des SCAT eine präzisere Auskunft und Kategorisierung über das Problem zu geben.

Was die einzelnen Erkennungsklassen beinhalten und welche Bedrohungen von Fehlern ausgehen, die dieser Klasse zuzuordnen sind, wird nun beschrieben.

Schwache Verschlüsselungsalgorithmen

Wie in der Einleitung bereits gezeigt wurde, ist die Verwendung veralteter Verschlüsselungsalgorithmen auch in der heutigen Zeit noch ein Problem. Rund 10%

[12] https://www.codeproject.com/Articles/26085/File-Encryption-and-Decryption-in-C, Zugriff: 15.12.2018

[13] https://cwe.mitre.org/data/definitions/310.html, Zugriff: 15.12.2018

aller untersuchten Anwendungen verwenden das Blockverschlüsselungsverfahren DES (Krüger et al., 2018; Egele et al., 2013). Die DES-Verschlüsselung ist jedoch anfällig für Brute-Force Angriffe (Diffie & Hellman, 1977). Je nach verfügbarer Rechenleistung, sind für einen erfolgreichen Angriff nur wenige Tage notwendig (Kumar et al., 2006). SCATs, die diese Erkennungsklasse umsetzten, müssen Kenntnis über schwache oder gebrochene Verfahren haben und die Verwendung dieser im Quelltext finden können.

Schwache Hashingalgorithmen

Hashfunktionen berechnen aus einer beliebig langen Eingabenachricht eine neue Nachricht mit festgelegter Länge (Wätjen, 2018, S. 93), dem Hashwert oder Fingerabdruck. Ziel dieser Hashfunktionen ist, dass verschiedene Eingabenachrichten niemals den selben Hashwert ergeben, die gleiche Nachricht aber immer den selben Wert erzeugt. Aus dem Hashwert darf sich die Eingabenachricht nicht wiederherstellen lassen. In der Praxis werden diese Hashwerte häufig benutzt, um das Schutzziel der Integrität zu überprüfen. Eine Bedrohung, die von einer Hashfunktion ausgeht, ist die Möglichkeit Kollisionen zu erzeugen. Bei einer Kollision liefert die Hashfunktion bei zwei verschiedenen Nachrichten den selben Hashwert. Hashingverfahren, bei denen bereits Kollisionen gefunden wurden, sind unter anderem SHA-1 (Wang, Yin & Yu, 2005) oder MD5 (Wang & Yu, 2005). Ein SCAT sollte in der Lage sein, die Verwendung dieser Verfahren im Quelltext zu finden.

Schwache Konfiguration der benutzten API

Krypto-APIs bieten den Programmierern eine Vielzahl von Verfahren und Konfigurationsmöglichkeiten an. Dank schnellerer Rechner oder bekanntgewordener Schwachstellen sind einige dieser Konfigurationsmöglichkeiten nicht mehr sicher genug, um die Sicherheit der Software zu gewährleisten. Ein bekanntes Beispiels ist die Verwendung des Betriebsmodus ECB für Blockverschlüsselungsverfahren. Mit diesem Modus werden alle Blöcke des Klartexts unabhängig voneinander verschlüsselt. Identische Blöcke produzieren daher auch identische Geheimtexte, wodurch die verschlüsselte Nachricht bezogen auch ihre Vertraulichkeit und Integrität angreifbar ist[14].

Verstoß gegen die API-Spezifikation

Diese Erkennungsklasse beschreibt Fehler, bei denen sich der Programmierer nicht an die Vorgaben der API hält. Laut Tsipenyuk et al., 2005 ist dieser Punkt sehr wichtig. Er setzt jedoch auch voraus, dass das SCAT die Konventionen und Verhaltensweisen der Krypto-API kennt. Darauf basierend kann das SCAT Aussagen treffen, ob der geschriebene Anwendungscode gegen die Vorgaben der API verstößt. Szenarien, die unter diese Klasse fallen, sind zum Beispiel, dass der Entwickler nach der Benutzung eines Verschlüsselungsverfahren vergisst, Speicher und Ressourcen der Verschlüsselungsinstanz, wie einem AES-Anbieter, zu löschen. Klassen, die das explizite Leeren erfordern, implementieren in C# üblicherweise die Schnittstelle `IDisposable`[15].

[14] Angriffsszenarien auf ECB: https://zachgrace.com/posts/attacking-ecb/, https://blogs.msdn.microsoft.com/varun_sharma/2007/11/27/block-ciphers-simple-attack-on-ecb-mode/, Zugriff jeweils: 16.12.2018

[15] https://docs.microsoft.com/de-de/dotnet/api/system.idisposable, Zugriff: 17.12.2018

Auslesbare Informationen

Eine große Bedrohung, die von Programmiersprachen den Programmiersprachen Java und C# ausgeht, ist die Möglichkeit ihre Programmdateien (`.jar`, `.exe` oder `.dll`) mit speziellen Werkzeugen zu dekompilieren, um den gesamten Quelltext in mehr oder weniger Originalform wiederherzustellen. Das in der Einleitung genannte Beispiel mit der Tagebuch-App demonstriert diese Verwundbarkeit. Passwörter, Schlüssel und andere Informationen können so von Angreifern ausgespäht werden. Besonders bei Client-Anwendungen, bei denen die Programmdateien lokal auf den Geräten der Anwender liegen und damit für einen Angreifer leicht zugänglich sind, stellen ein hohes Risiko dar. Ein SCAT sollte daher in der Lage sein zu erkennen, ob geheime Daten im Quelltext kodiert sind.

4.4.2 Abgrenzung des Funktionsumfangs

Krypto-APIs bieten ein vielfältiges Angebot von kryptographischen Funktionen an. Die Betrachtung und Integration aller möglichen Anwendungsfehler einer API durch Sharper Crypto-API Analysis ist nicht realistisch. Der Funktionsumfang soll sich daher auf einige Bereiche konzentrieren. Durch die Anforderungen **F5** kann überdies sichergestellt werden, dass in Zukunft hier nicht berücksichtigte Funktionen in das SCAT aufgenommen werden können. Unter dem Funktionsumfang bzw. Funktionsangebot werden in diesem Abschnitt lediglich die einzelnen verfügbaren Analyzers und Codegenerierungsaufgaben der Nutzung von Krypto-APIs verstanden.

Bei der Entscheidungsfindung, welche Codeanalysen und Codegenerierungsaufgaben in diesem SCAT umgesetzt werden sollen, wurden zwei Kriterien berücksichtigt:

1. Vorkommen der Aufgabe
2. Geschätzter Umsetzungsaufwand

Das erste Kriterium lässt sich anhand der Tabelle 3.1 bestimmen. Aus ihr geht hervor, dass die symmetrische Verschlüsselung die am häufigsten benötigte Aufgabe ist. Sie wird daher gegenüber der asymmetrischen Verschlüsselung für die Entwicklung von Sharper Crypto-API Analysis priorisiert in Betracht gezogen.

Um zu zeigen, dass sich dieses SCAT jedoch nicht nur auf die symmetrische Verschlüsselung begrenzen soll, wurde eine weitere Aufgabe gewählt, die nach dem zweiten Kriterium einen möglichst geringen Umsetzungsaufwand erfordern. Grund für dieses Kriterium ist der begrenzte Zeitrahmen dieser Arbeit. Mit knapp einem Viertel zählt die Generierung geheimer Schlüssel, wie sie auch bei passwortbasierten Verschlüsselungsverfahren benötigt werden, noch zu den drei häufigsten Krypto-Aufgaben. Zudem konnte basierend auf den bereits gesammelten Erfahrungswerten ein geringer Umsetzungsaufwand für Funktionen in dieser Kategorie geschätzt werden. Die Generierung von Schlüsseln, insbesondere für passwortbasierte Verfahren, wird daher dem Funktionsumfang von Sharper Crypto-API Analysis hinzugefügt.

4.4.3 Funktionsangebot der .NET Krypto-API

Um nun Analyzer für Sharper Crypto-API Analysis vorschlagen und entwickeln zu können, müssen zunächst die verfügbaren Funktionalitäten der API aufgezeigt wer-

den. Dieser Abschnitt gibt dem Leser einen groben Überblick über vorhandene Algorithmen und Klassen der Krypto-API.

Die .NET Krypto-API ist unter dem Namespace `System.Security.Cryptography` verfügbar. Weitere Namespaces wie `System.Security` werden in dieser Arbeit nicht betrachtet. Eine vollständige Dokumentation der API ist auf der Webseite von Microsoft, dem Entwickler der API, bereitgestellt[16]. Beim Besuch dieser Seite fällt unter anderem auf, dass der Nutzer zwischen den verschiedenen .NET Plattformen auswählen kann. Jede einzelne Plattform bietet eine ähnliche, wenn auch nicht identische Sammlung an Schnittstellen und Typen an, die für diese Plattform verfügbar sind. So beinhaltet das .NET Framework beispielsweise auch Klassen, deren Implementation auf native Funktionsbibliotheken des Betriebssystems Windows zurückgreifen. Die betriebssystemunabhängige Plattform .NET Core besitzt diese Klassen hingegen nicht. Um diese Konflikte zu vermeiden, wird sich die Arbeit nur auf die verfügbaren Kassen der .NET Standard 2.0 API-Spezifikation beziehen.

Grundsätzlich lässt sich das Funktionsangebot der .NET Krypto-API in fünf verschiedene Funktionsgruppen unterteilen:

1. Symmetrische Verschlüsselungsverfahren
2. Asymmetrische Verschlüsselungsverfahren
3. Hashfunktionen
4. Zufallsgeneratoren
5. Schlüsselableitungsfunktionen

Weitere spezielle Datentypen helfen nur der Implementierung der oben genannten Funktionsgruppen und werden daher nicht separat als eigene Gruppe aufgeführt. Des Weiteren wird in diesem Abschnitt auf eine weitere Erklärung von verfügbaren asymmetrischen Verschlüsselungsverfahren und ihren Besonderheiten aufgrund der bereits festgelegten Abgrenzung des Funktionsumfangs von Sharper Crypto-API Analysis abgesehen.

Verfügbare Symmetrische Verschlüsselungsverfahren

Neben den bereits häufig angesprochenen DES und AES Verfahren, bietet die .NET Krypto-API noch weitere symmetrische Verschlüsselungsverfahren an. Die Namen in der folgenden Auflistung stehen dabei für den Klassennamen, die dieses Verfahren beinhalten.

- `AES`
- `DES`
- `RC2`[17]

[16] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography, Zugriff: 17.12.2018

[17] RC2 (Rivest, 1998) ist ein Blockverschlüsselungsverfahren, welches durch Related-Key Angriffe verwundbar ist (Kelsey, Schneier & Wagner, 1997).

- `Rijndael`[18]
- `TripleDES`

Die abstrakte Basisklasse dieser Verfahren trägt den Namen `SymmetricAlgorithm`. Die einzelnen Verschlüsselungsverfahren werden darüber hinaus in verschiedenen Implementationen bereitgestellt. Unterschieden wird hier zwischen Klassen, die in C# programmiert wurden und Wrappern, die auf die Krypto-API des Betriebssystems zugreifen. Erstere werden durch den Zusatz `Managed` im Klassennamen gekennzeichnet, wie zum Beispiel `AesManaged`. Die Wrapper haben den Klassennamenzusatz `CryptoServiceProvider`.

Verfügbare Hashfunktionen

Die .NET Krypto-API bietet verschiedene kryptographische Hashfunktionen an, die alle unter der abstrakten Basisklasse `HashAlgorithm` vereint sind. Zu den verfügbaren Algorithmen gehören:

- **MD5**,
- **RIPEMD-160**[19],
- **SHA-1**,
- **SHA-2** in den Hashwertlängen: 256, 384, 512 Bits,
- sowie eine **Keyed-Hash Message Authentication Code (HMAC)** Implementierung aller bisherigen Hashfunktionen.

Analog wie bei den symmetrischen Verschlüsselungsverfahren, werden einige Hashfunktionen, darunter SHA1 und SHA2, in den verschiedenen Varianten `Managed` und `CryptoServiceProvider` unterstützt.

Verfügbare Schlüsselableitungsfunktionen

Schlüsselableitungsfunktionen generieren anhand einer Eingabe, dem Ableitungsschlüssel (beispielsweise einem Passwort), und einem Pseudozufallsgenerator einen oder mehrere Schlüssel (Chen, 2008). In der passwortbasierten Schlüsselableitung werden als weitere Eingabeparameter ein Salt und Kostenfaktor angegeben (Kaliski, 2000). Das Salt wird dem Passwort hinzugefügt, damit es einzigartiger wird. Der Kostenfaktor bestimmt den benötigten Aufwand der Ableitungsfunktion. Durch diese zwei Maßnahmen werden die generierten Schlüssel schwerer angreifbar für Wörterbuchangriffe (Morris & Thompson, 1979), Rainbow Tabellen (Oechslin, 2003) und schnelleren Computern.

[18] Die Klasse `Rijndael` implementiert in gewisser Maßen das AES Verfahren, nur ohne die Einschränkung einer festen Blockgröße von 128 Bits. Bei diesem Verfahren kann die Blockgröße zwischen 128 und 256 Bits und allen Zahlen in 32er Schritten dazwischen gewählt werden.

[19] Beim RIPEMD-160 handelt es sich im eine eher unbekannte und wenig verbreite Hashfunktion, die kaum untersucht wurde. 2006 konnte jedoch gezeigt werden, dass sie gegen bekannte Angriffe sicher ist (Mendel, Pramstaller, Rechberger & Rijmen, 2006).

Die .NET Krypto-API stellt eine abstrakte Klasse namens `DeriveBytes` bereit, die für diese Funktion zuständig ist. Von ihr erben die Klassen `PasswordDeriveBytes` und `Rfc2898DeriveBytes`. Erstere implementiert das Password-Based Key Derivation Function (PBDKF)1 Verfahren und zweitere das PBDKF2 Verfahren. Bei der Implementation des PBDKF1 Verfahren handelt es sich jedoch um eine Erweiterung durch Microsoft[20].

Verfügbare Zufallsgeneratoren

Der Zufall spielt in der Kryptographie eine wichtige Rolle. Wie in der Einleitung bereits beschrieben, muss der IV bei Blockverschlüsselungsverfahren die Eigenschaft besitzen, sowohl zufällig, als auch unvorhersehbar sowie einzigartig (Dworkin, 2001, S. 8) zu sein. Dies trifft ebenfalls auf das Salt bei den Schlüsselableitungsfunktionen zu (Turan, Barker, Burr & Chen, 2010, S. 6). Die Verwendung des Pseudozufallsgenerators der Klasse `System.Random` kann daher zu diesem Zweck nicht verwendet werden, da die Generierung der Zufallswerte von einem Startwert, dem Seed, abhängt. Dieser Startwert kann unter anderem die aktuelle Systemzeit sein. Ist dieses Seed bekannt, können alle generierten Werte rekonstruiert werden und sind somit vorhersehbar. Kryptographisch sichere Pseudozufallsgeneratoren haben die besondere Eigenschaft, pseudo-zufällige Werte zu generieren, die sich von echt-zufälligen Werten nicht unterscheiden lassen (Santha & Vazirani, 1986). Die .NET Krypto-API bietet zu diesem Zweck die Klasse `RNGCryptoServiceProvider` an. Intern greift diese Klasse auf die nativen Krypto-APIs der jeweiligen Betriebssysteme zurück. In Windows wird das zufällige Generieren von Zahlen beispielsweise durch die Assembly `bcrypt.dll` durchgeführt.

4.4.4 Beschreibung der Analyzers in Sharper Crypto-API Analysis

Dieser Abschnitt stellt die Analyzer vor, die im Rahmen der Masterarbeit geplant sind. Ihre Auswahl geht einerseits aus der untersuchten Literatur und den Programmierernetzwerken wie Stackoverflow oder Codeproject und andererseits aus der oben beschriebenen Abgrenzung des Funktionsumfangs hervor.

Die Tabelle im Anhang B zeigt alle geplanten Analyzer. Zu den Angaben gehören eine Analyse-ID, die auch in der IDE als Identifikation der Codeanalyse genutzt wird. Die Spalten „Bezeichnung“ und „Beschreibung“ geben eine kurze schriftliche Auskunft, unter welchen Umständen die Analyse gemeldet wird. Ob der Analyzer zum Zeitpunk der Abgabe dieser Arbeit implementiert ist, gibt die Spalte „Status“ an. Eine Einordnung des Analyzers in einer der oben beschriebenen Erkennungsklassen erfolgt in der Spalte „Erkennungsklasse“

In ihrer Arbeit zur Untersuchung von häufigen Fehlern bei der Anwendung von Krypto-APIs in Android Apps listen Egele et al. jeweils mit Begründung sechs Regeln auf, die beim Verwenden der JCA zu beachten sind:

Regel 1: Verwende nicht den Modus ECB für Verschlüsselungen.

[20] s. Remarks https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.passwordderivebytes, Zugriff: 17.12.2018

> Regel 2: Verwende nicht einen nicht-zufälligen IV für CBC Verschlüsselungen.
>
> Regel 3: Verwende keine konstanten Verschlüsselungsschlüssel.
>
> Regel 4: Verwende kein konstantes Salt für PBE [Anm. d. Verf.: password based encryption].
>
> Regel 5: Verwende nicht weniger als 1000 Interationen für PBE.
>
> Regel 6: Verwende keine statischen Seeds [...].
>
> Egele et al., 2013, S. 75

All diese Regeln, mit Ausnahme der Sechsten[21], sind ebenfalls für die Verwendung von C# Analysen anwendbar, da sie sich nicht explizit an die JCA richten, sondern allgemeingültig sind. Sie werden daher für die Entwicklung von Sharper Crypto Analysis übernommen. Diese und weitere implementierte Analyzer werden in den folgenden Unterabschnitten kurz erläutert.

Analyzer bezüglich Verschlüsselungsverfahren

Krypto-APIs bieten schwache Verfahren oder Konfigurationen an, Dokumentationen und Tutorials verwenden diese und der Programmierer implementiert sie in letzter Konsequenz in sein Programm. Welche Verfahren und Konfigurationsmodi in Sharper Crypto-API Analysis als Schwäche erkannt werden sollen, geht hauptsächlich aus den Richtlinien des National Institute of Standards and Technology (NIST) hervor (E. B. Barker, 2016; Dworkin, 2001; E. Barker, Feldman & Witte, 2017). Der Einsatz der Verschlüsselungsverfahren DES und Triple Data Encryption Standard (TDES) wird in diesen Dokumenten nicht empfohlen und wird daher vom SCAT erkannt. Des Weiteren erkennt das SCAT den durch Related-Key Angriffen verwundbaren RC2 Algorithmus (Kelsey et al., 1997) als Fehler. Listing 4.2 zeigt die Instanziierung der zwei symmetrischen Verschlüsselungsverfahren DES und TDES. Zu erkennen ist, dass die .NET Krypto-API sowohl die Instanziierung durch den Konstruktor, als auch durch Aufruf einer Factory-Methode (`Create`) erlaubt. Der Analyzer muss in der Lage sein, beide Fälle richtig zu bewerten. Die Standardwarnstufe, die der Analyzer ausgibt, ist eine Warnung. Dieser Analyzer besitzt die ID **SCAA008**.

```
1 var tdes = new TripleDESCryptoServiceProvider();
2 var des = SymmetricAlgorithm.Create("DES");
```

Listing 4.2: Instanziierung von symmetrischen Verschlüsselungsverfahren in C#

Als verfügbare symmetrische Verschlüsselungsverfahren in der .NET Krypto-API bleiben demnach `AES` und die allgemeinere Form `Rijndael` zur sicheren Verwendung. Die API stellt fünf verschiedene Betriebsmodi (CBC, ECB, OFB, CTS, CFB) bereit. Von ihnen bietet ECB eine große Angriffsfläche. Während der Vorteil eine hohe Parallelisierbarkeit darstellt, ist der Nachteil, dass alle Blöcke mit gleichem Klartext auch denselben Geheimtext erzeugen und dadurch angreifbar ist[22][23]. Der Analyzer muss in

[21] In der .NET Krypto-API kann kein Seed zum Berechnen einer Zufallszahl angegeben werden.

[22] Beschreibung eines Chosen Plaintext Angirffs: https://zachgrace.com/posts/attacking-ecb/, Zugriff: 25.12.2018

[23] Verletzung der Integrität einer verschlüsselten Nachricht: https://blogs.msdn.microsoft.com/varun_sharma/2007/11/27/block-ciphers-simple-attack-on-ecb-mode/, Zugriff: 25.12.2018

der Lage sein die Verwendung dieses Betriebsmodus', wie in Listing 4.3 dargestellt, zu erkennen. Die Standardwarnstufe beim Verwenden von ECB ist ein Fehler. Dieser Analyzer besitzt die ID **SCAA009**.

```
1 using (var aes = new AesCryptoServiceProvider())
2 {
3     aes.Mode = CipherMode.ECB;
4 }
```

Listing 4.3: Verwnedung von AES mit dem Betriebsmodus ECB in C#

Analyzer bezüglich des IV oder Verschlüsselungsschlüssel

Neben den häufig eingesetzten schwachen Verschlüsselungsverfahren und Betriebsmodus ECB, werden viele Fehler im Umgang mit dem IV oder Schlüssel begangen. Darunter auch, wie im Beispiel der Tagebuch App aus der Einleitung bereits aufgeführt, das Hartkodieren dieser Daten im Quelltext, was den oben genannten Regeln zwei und drei widerspricht. Die Analyzer in Sharper Crypto-API Analysis müssen feststellen können, ob zu Zuweisung des IV und Schlüssels anhand von konstanten Werten stattfindet. Da dieses Tool ein SCAT darstellt, kann jedoch nur überprüft werden, ob die Daten nur zum Zeitpunkt des Kompilierens in die CIL konstant sind. In Listing 5.1 wird das Zuweisen des IV und Schlüssels dargestellt. Zu erkennen ist, dass die Variable *ivAndKey* eine konstante Wertezuweisung besitzt. Sowohl IV, als auch Schlüssel beziehen ihre Daten indirekt über diese Variable. Der Analyzer muss also in der Lage sein, den Datenfluss der Zuweisungen (Zeilen 6 & 7) auf die Variablendeklaration in Zeile 1 zurückzuverfolgen. Die Standardwarnstufen für diese Codeanalysen sind im Falle eines konstanten IV eine Warnung und im Falle eines konstanten Schlüssel ein Fehler. Diese Analyzers besitzen die IDs **SCAA001** und **SCAA002**.

```
1 var ivAndKey = new[] {(byte) 123, (byte) 123, (byte) 123};
2 var iv = ivAndKey;
3 var key = ivAndKey;
4 using (var aes = new AesCryptoServiceProvider())
5 {
6 aes.IV = iv;
7 aes.Key = key;
8 }
```

Listing 4.4: Setzten eines konstanten IV und Schlüssel für AES in C#

Des Weiteren ist dem Listing 5.1 zu entnehmen, dass IV und Schlüssel den selben Wert annehmen. Da der IV jedoch öffentlich bekannt sein muss, um eine Entschlüsselung durchzuführen[24], hat dies zur Folge, dass auch der Schlüssel, der geheim sein muss, verraten wird. Die Verschlüsselung ist in diesem Fall obsolet. Der Analyzer muss also in der Lage sein zu erkennen, ob die Zuweisungen für den IV und Schlüssel den selben Ursprung haben, voneinander abhängig sind oder den gleichen, konstanten

[24]Nicht jeder Betriebsmodus, wie auch ECB, verwendet einen IV, sodass dieser nicht automatisch für jede Entschlüsselung benötigt wird.

Wert besitzen. Die Standardwarnstufe dieses Analyzers ist ein Fehler. Dieser Analyzer besitzt die ID **SCAA003**.

Analyzer bezüglich einer passwortbasierten Verschlüsselung

Bei passwortbasierten Verschlüsselungsverfahren wird der geheime Verschlüsselungsschlüssel aus einem Passwort mittels einer Schlüsselableitungsfunktion wie PBKDF2 generiert [25]. Die verfügbaren Parameter der Funktion, neben dem Passwort, sind das Salt (vom Typ `byte[]` bzw. `int`) und die Anzahl der Berechnungsiterationen (vom Typ `int`). Das Listing 4.5 zeigt beispielhaft eine schwache Verwendung der Schlüsselableitungsfunktion. Die Analyzer in Sharper Crypto-API Analysis müssen in der Lage sein, die Schwächen in diesem Listing zu finden.

Ähnlich wie bei den Analyzern **SCAA001** und **SCAA002** fällt hier zunächst auf, dass das abzuleitende Passwort in der Variable `pw` hartkodiert ist und daher einfach durch das Dekompilieren des Codes herauszufinden ist. Ein Analyzer muss dieses Vergehen auffinden können. Die Standardwarnstufe ist ein Fehler. Des Weiteren soll dieser Analyzer noch auf konstante Variablendeklarationen achten, bei welchen der Variablenname darauf hinweisen könnte, dass ein Passwort hartkodiert wurde. Einige dieser verdächtigen Namen sind: `password`, `pw` oder `pass`. Die Standardwarnstufe für diesen Fall ist eine Mitteilung. Dieser Analyzer besitzt die ID **SCAA005**.

Eine weitere Schwäche in Listing 4.5 ist die Anzahl der Iterationen, die die Ableitungsfunktion durchführen soll. Zum Zeitpunkt, als die Funktion PBKDF2 im Jahr 2000 veröffentlicht wurde (Kaliski, 2000), lag die empfohlene Anzahl der Iterationen bei 1.000, dem Standardwert in der .NET Krypto-API. Derzeit wird vom NIST eine Mindestanzahl von 10.000 Iterationen vorgeschlagen (Grassi et al., 2017, S. 15). Der Analyzer muss daher bewerten können, ob die vorgegebene Anzahl der Iterationen im Quellcode kleiner als die Empfehlung des NIST ist. Die Standardwarnstufe für diesen Analyzer ist eine Warnung. Dieser Analyzer besitzt die ID **SCAA011**.

Beim Umgang mit dem Salt sind in Listing 4.5 zwei Schwächen festzustellen. Gemäß den Richtlinien des NIST soll das Salt eine Mindestlänge von 16 Bytes (128 Bits) besitzen. Zudem soll das Salt aus willkürlich generierten Werten bestehen (Turan et al., 2010, S. 6). Der gegebene Code erfüllt beide Anforderungen nicht. Das Salt ist nicht 16, sondern nur drei Bytes lang und seine Werte sind nicht willkürlich. Hinzu kommt, dass dasselbe Salt für jede einzelne Schlüsselableitung mit diesem Code verwendet wird, wodurch die generierten Schlüssel der Anwendung für Rainbow Tabellen angreifbar werden. Die Standardwarnstufe für diese Analyzer ist eine Warnung. Sie besitzen die IDs **SCAA010** und **SCAA012**.

```
1 var salt = new byte[] {123, 123, 123};
2 var pw = "123456";
3 using (var pbkd2 = new Rfc2898DeriveBytes(pw, salt, 1000))
4 {
5 }
```

Listing 4.5: Schwache Verwendung der Schlüsselableitungsfunktion PBKDF2 in C#

[25] Auf das unmittelbare Verwenden des Passworts oder dem Hashwert des Passworts als geheimer Schlüssel sollte aufgrund der schnellen Berechenbarkeit sämtlicher Schlüssel (Brute Force Attack) oder schnellen Generierung und Abgleichung von Wörterbüchern und Rainbow Tabellen (Oechslin, 2003) vermieden werden.

Weitere Analyzer

Neben den bisher aufgeführten Analyzern wurden noch drei weitere Analyzer dem Funktionsumfang von Sharper Crypto-API Analysis hinzugefügt.

In vielen Fällen, so auch beim Salt oder IV ist die Einzigartigkeit und Zufälligkeit eine wichtige Anforderung. Wie bereits in Abschnitt 4.4.3 geschildert, eignet sich der Einsatz der Klasse `System.Random` nicht, um diese Anforderungen sicherzustellen. Ein Analyzer soll daher bei der Verwendung dieser Klasse einen Hinweis geben können. Um die False Positive Rate niedrig zu halten, wird jedoch in den Standardeinstellungen keine Warnstufe in der IDE angezeigt. Dieser Analyzer besitzt die ID **SCAA004**.

Die .NET Krypto-API gibt dem Programmierer einige Hashfunktionen zur Auswahl. Standardmäßig wird von der API das SHA-1 Verfahren bevorzugt. Das NIST stellt eine Liste bereit, in welcher Hashfunktionen und ihre für genehmigten Anwendungsbereiche aufgeführt werden (E. B. Barker & Roginsky, 2015, S. 14)[26]. Das Hashingverfahren SHA-1 wird in diesem Dokument für nur einige Anwendungsgebiete zugelassen, darunter zur Herstellung von Kompatibilität alter Daten oder der Verwendung von Schlüsselableitungsfunktionen. Zur Signierung von Daten oder Zeitstempeln soll das Verfahren jedoch nicht mehr genutzt werden. Ein Analyzer in Sharper Crypto-API Analysis soll daher in der Lage sein, die Verwendung von Hashingverfahren im Anwendungskontext zu erkennen und gegebenenfalls eine Meldung bereitstellen. Demnach soll der Analyzer die Nutzung von SHA-1 beim PBKDF2 Verfahren erlauben. Die Standardwarnstufe dieses Analyzers ist eine Mitteilung. Dieser Analyzer besitzt die ID **SCAA007**.

Einige Anwendungen, wie Informationssysteme von Regierungen, erfordern die Nutzung zertifizierter kryptographischer Module. Ein solches Modul kann beispielsweise die Implementierung eines symmetrischen Verschlüsselungsverfahrens sein. Die Zertifizierung erfolgt nach dem Standard FIPS PUB 140-2[27]. Die .NET Krypto-API bietet dem Programmierer mehrere Implementationen seiner Verschlüsselungs- und Hashingverfahren an. Sie lassen sich in die sogenannten „managed“ und nativen Implementierungen einteilen. Während die managed Funktionen innerhalb der .NET Plattform implementiert sind, greifen die nativen Funktionen mittels Wrapper auf die Implementierungen der Betriebssystemmodule zurück. Diese Betriebssystemmodule sind in der Regel nach FIPS PUB 140-2 zertifiziert[28]. Ein Programmierer, der die .NET Krypto-API nutzt, kann anhand des Klassennamens erkennen, ob die verwendete Implementierung managed oder nativ ist. Managed Implementierungen tragen das Suffix `Managed`, während die Betriebssystemwrapper das Suffix `CryptoServiceProvider` oder `Cng` besitzen. Ein Analyzer in Sharper Crypto-API Analysis soll erkennen können, ob der Programmierer eine zertifizierte Implementierung nutzt, oder nicht. Die Standardeinstellungen sehen vor, dass dieser Analyzer keine Warnstufe an die IDE meldet. Dieser Analyzer besitzt die ID **SCAA006**.

[26] Die Genehmigung eines kryptographischen Verfahrens durch das NIST basiert auf den Anforderungen für US-Regierungsbehörden zum Schutz von Informationen mit der Einstufung: Sensitive But Unclassified (E. B. Barker & Roginsky, 2015, S. 1).

[27] https://csrc.nist.gov/publications/detail/fips/140/2/final, Zugriff: 26.12.2018

[28] Liste von zertifizierten Krypto-Modulen in Windows: https://docs.microsoft.com/en-us/windows/security/threat-protection/fips-140-validation, Zugriff: 30.12.2018

4.5 Programmierung

Dieser Abschnitt befasst sich mit der Dokumentation der Entwicklung von Sharper Crypto-API Analysis. Während sich die Abschnitte 4.3 und 4.4 mit den Funktionsbeschreibungen der Tools befasst haben, werden hier auf die technischen Details eingegangen, die diese Funktionen realisieren. Dazu gehören auch Erklärungen von verwendeten Konzepten sowie ein tieferer Einblick in die Implementierung einzelner Komponenten. Zunächst gibt der Abschnitt 4.5.1 einen kurzen Überblick über den Aufbau des Git Repository, auf welchem der Quellcode von Sharper Crypto-API Analysis verfügbar ist. Danach folgt in Abschnitt 4.5.2 ein grober Überblick über die vorhandene Architektur des SCAT. Die Abschnitte 4.5.3 bis 4.5.6 beschreiben die einzelnen Funktionskomponenten und deren technische Umsetzung. Abschließend erläutert Abschnitt 4.5.7 kurz den korrekten Umgang von Sharper Crypto-API Analysis.

4.5.1 Git Repository

Bevor Details aus der Programmierphase des SCAT berichtet werden, wird hier zur besseren Orientierung zunächst ein Überblick über den Aufbau des verwendeten Git Repository geschafft.

Sharper Crypto-API Analysis wurde während der Bearbeitungsphase dieser Masterarbeit auf dem hochschulinternen GitLab-Server entwickelt. Mit Abschluss der Arbeit erfolgt eine Portierung des Codes auf ein öffentliches GitHub Repository[29] dieses Autors. Der Aufbau beider Repositorys ist identisch. Nach Abschluss der Arbeit wird zur weiteren Entwicklung ausschließlich das öffentliche GitHub Repository benutzt.

Der gesamte Quellcode beschränkt sich auf die folgenden Unterordner:

\src: Beinhaltet Code der Visual Studio Erweiterung und der Codeanalysen.

\tests: Beinhaltet Modultests zu vereinzelten Komponenten sowie zu allen Codeanalysen.

\tool: Beinhaltet Beispiele zum Umgang mit dem von Sharper Crypto-API Analysis bereitgestellten SDK sowie ein Hilfsprogramm zu Einrichtung von Softwareprojekten (s. Abschnitt 4.5.7).

Die Ordnerstruktur dieses Repository wurde dem von Microsoft verwendeten Aufbau eines Repository entnommen[30].

4.5.2 Architektur

Projektstruktur

Die Struktur von .NET Programmen ist unter anderem anhand der programmierten Assemblies zu erklären.

[29] Link zu Sharper Crypto-API Analysis: https://github.com/AnakinSklavenwalker/SharperCryptoApiAnalysis, Zugriff: 13.12.2018

[30] z.B. https://github.com/dotnet/coreclr, Zugriff: 13.12.2018

Eine Assembly im Sprachgebrauch der .NET Programmierung ist eine Programmkomponente, zum Beispiel eine Funktionsbibliothek, die in als `.exe` oder `.dll` Datei vorliegt. Sie bildet eine logische Einheit. .NET Programme können auf diese Assemblies referenzieren, um somit die darin enthaltenden Klassen, Typen und Ressourcen zu nutzen[31].

Die Projektstruktur von Sharper Crypto-API Analysis lässt sich in drei Bereiche gliedern:

1. Visual Studio Plugin
2. Analyzers (Erweiterungen)
3. Verbindungscode (SDK)

Um diese Gliederung erklären zu können, muss zunächst verstanden werden, dass Plugins und Analyzers in Visual Studio grundsätzlich zwei verschiedene Dinge darstellen. Plugins oder Programmerweiterungen für Visual Studio sind Softwarepakete, die das Verhalten und Aussehen der IDE verändern und erweitern können. Sie greifen dazu auf das Visual Studio SDK[32] zu. Dem Programmierer eines Plugins stehen so die Möglichkeiten zur Verfügung, auf den Texteditor zuzugreifen, Menüleisteneinträge zu bearbeiten oder neue Toolfenster[33] zu entwickeln. Installierte Plugins werden mit dem Starten von Visual Studio geladen. Analyzer sind eigenständige Assemblies und enthalten zunächst einmal nur Routinen, die den Quelltext auf bestimmte Eigenschaften überprüfen und korrigieren können. Es gibt zwei Möglichkeiten Analyzer in Visual Studio zu integrieren. Die erste Möglichkeit ist das Verwenden eines Plugins. Das Plugin referenziert einen oder mehrere Analyzer. Das Plugin sowie die Analyzer werden mit dem Starten von Visual Studio geladen. Durch interne Mechanismen werden die Analyzers beim Öffnen von Quelltextdateien automatisch ausgeführt. Da Analyzers jedoch eigenständige Assemblies sind, also `.dll` Dateien, können Softwareprojekte jedoch auch direkt auf diese Assembly referenzieren. Ein Plugin ist dadurch nicht erforderlich. Visual Studio erkennt, dass die Referenzierte Assembly Codeanalysen besitzt und führt diese dann automatisch aus.

Der erste Bereich der oben genannten Projektstruktur (Visual Studio Plugin) enthält lediglich die Assemblies, die direkt mit der IDE interagieren können. Die Namen und Funktionsart dieser Assemblies werden im folgenden aufgelistet:

> `SharperCryptoApiAnalysis.dll`. Diese Assembly enthält den Einstiegspunkt, mit dem Visual Studio erkennen kann, dass die Assembly ein Plugin bereitstellt. Hier werden Menüeinträge und Toolfenster implementiert. Der Code ist unter dem besonderen Pfad `\src\SharperCryptoApiAnalysis.Vsix\`[34] im Git Repsitory zu finden.

[31] https://msdn.microsoft.com/de-de/library/bb978926.aspx, Zugriff. 13.12.2018

[32] https://docs.microsoft.com/en-us/visualstudio/extensibility/visual-studio-sdk, Zugriff: 13.12.2018

[33] Die genaue Spezifizierung eines Toolfenster liefert Microsoft hier: https://docs.microsoft.com/en-us/visualstudio/extensibility/extending-and-customizing-tool-windows, Zugriff: 13.12.2018

[34] Das Suffix *Vsix* soll kennzeichnen, dass dieses Projekt den Einstiegspunkt des Plugin enthält. Eine `.vsix` Datei ist üblicherweise die Installationsdatei eines Plugins für Visual Studio.

`SharperCryptoApiAnalysis.Shell.dll`. Diese Assembly enthält unter anderem eine Implementation des Model View ViewModel (MVVM) Pattern und implementiert Kontrollelemente.

`SharperCryptoApiAnalysis.Extensibility.dll`. Diese Assembly enthält Implementationen, welche die Erweiterbarkeit und Konfigurierbarkeit des Plugins ermöglichen (s. Abschnitt 4.5.5).

`SharperCryptoApiAnalysis.Connectivity.dll`. Diese Assembly implementiert die Verbindung des Plugins zu Git-Diensten wie GitHub (s. Abschnitt 4.5.5).

Der zweite Bereich der Projektstruktur (Analyzers) enthält Assemblies, die die Roslyn-basierten Analyzer bereitstellen. Derzeit existiert nur die Assembly `SharperCrypto-ApiAnalysis.BaseAnalyzers.dll`. In ihr sind alle in dieser Arbeit entwickelten Analyzer enthalten. Im Repository ist dieses Projekt unter dem Pfad `\src\Extensions\` zu finden. In Sharper Crypto-API Analysis werden Analyzererweiterungen allgemein als *Extension* bezeichnet.

Der letzte Bereich beinhaltet Assemblies, die dafür sorgen, dass sowohl Plugin, als auch Extensions miteinander kommunizieren können. Sie stellen dazu Schnittstellen und Basisimplementationen bereit. Diese Assemblies stellen gleichermaßen das SDK von Sharper Crypto-API Analysis dar:

`SharperCryptoApiAnalysis.Core`: Diese Assembly enthält Konstanten und rudimentäre Implementierungen.

`SharperCryptoApiAnalysis.Interop.dll`. Diese Assembly enthält zusätzliche Schnittstellen und Basisimplementationen zur Entwicklung von Extensions. Die wichtigsten Komponenten sind das Bereitstellen einer Template Engine (s. Abschnitt 4.5.4), Interaktionsschnittstellen zur Konfigurierung des SCAT (s. Abschnitt 4.5.5) sowie erforderliche Komponenten zum Erstellen von Codeanalysen und Analyseberichten (s. Abschnitt 4.5.3).

`SharperCryptoApiAnalysis.Shell.Interop.dll`: Diese Assembly enthält Basisimplementationen und Schnittstellen zur Entwicklung und Nutzung von Benutzeroberflächen. Darunter fällt auch die Implementation und Bereitstellung eines graphischen Benutzerassistenten. Ebenfalls bietet diese Assembly Komponenten zur Erstellung von Codegenerierungsaufgaben (s. Abschnitt 4.5.4) an.

`SharperCryptoApiAnalysis.VisualStudio.Integration.dll`: Diese Assembly abstrahiert einige Funktionen, darunter auch das Projekt- und Verwaltungssystem, von Visual Studio.

Das Zusammenspiel dieser Assemblies miteinander und der Interaktion mit Visual Studio zeigt die Abbildung 4.3. Daraus wird deutlich, dass das Plugin mit seinen Extensions nicht direkt, sondern nur über das SDK kommuniziert. Grund hierfür ist, dass die Architektur von Visual Studio diese Interkommunikation zwischen Plugins und Analyzers nicht direkt vorsieht. Das SDK von Sharper Crypto-API Analysis muss hier eine Brücke schaffen.

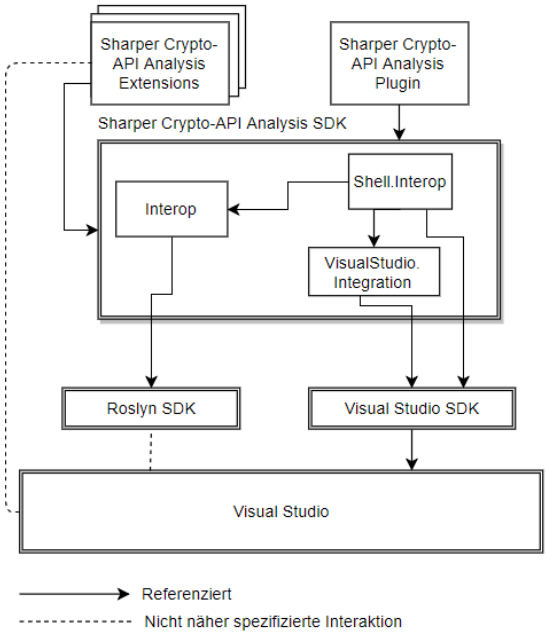

Abbildung 4.3: Projektabhängigkeiten in Sharper Crypto-API Analysis (Eigene Darstellung)

4.5.3 Analyzers

Codeanalyse durch Roslyn

Die Funktion der Codeanalyse wird nicht durch Sharper Crypto-API Analysis implementiert, sondern durch die Verwendung des Roslyn SDK. Mittels diesem SDK kann Quellcode nicht nur auf einer syntaktischen Ebene, sondern auch auf einer semantischen Ebene untersucht werden. Dadurch ist es beispielsweise möglich, den Datenfluss einer Variable nachzuvollziehen[35]. Eine Beschreibung, wie Roslyn-Analysen programmiert werden können, soll diese Arbeit jedoch nicht geben.

Das Roslyn SDK bietet zur Codeanalyse eine abstrakte Klasse namens `DiagnosticAnalyzer` an. Um weitere Funktionalitäten, wie die Interkommunikation von Codeanalysen mit dem Plugin Sharper Crypto-API Analysis zu ermöglichen, wurde sie durch die Klasse `SharperCryptoApiAnalysisDiagnosticAnalyzer` erweitert. Alle Codeanalysen für Sharper Crypto-API Analysis sollten daher von dieser neuen Klasse erben. Im aktuellem Entwicklungsstand sind die folgenden Interkommunikationsmöglichkeiten vorhanden:

- **Registrierung bei einer Verwaltungsinstanz:** In Sharper Crypto-API Analysis ist unter der Schnittstelle `IAnalyzerManager` ein Verwalter für alle Analyzer verfügbar. Derzeit können hier nur Analyseberichte bereitgestellt werden, doch weitere Funktionalitäten sind möglich.
- **Anpassung der Analysehärte:** Der Analyzer hat Zugriff auf die Benutzereinstellungen des SCAT. Zu diesen Einstellungen gehört auch die Analysehärte. Die Analysehärte bestimmt, wie mit welcher Warnstufe ein Analysebericht gemeldet werden soll. Jeder Analyzer hat die Möglichkeit durch die

[35] https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Semantic-Analysis, Zugriff: 15.12.2018

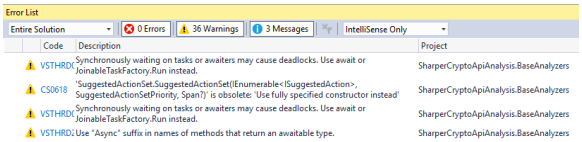

Abbildung 4.4: Fehlerfenster in Visual Studio

überschreibbare Methode `AnalysisSeverityToDiagnosticSeverity` ein Mapping zwischen der Analysehärte und der Warnstufe der IDE festzulegen. Wird die Einstellung der Analysehärte geändert, ändert sich auch der Analysebericht nach der Ausführung der nächsten Codeanalyse[36].

- **Bereitstellen von Analyseberichten:** Analyseberichte stellen in Sharper Crypto-API Analysis eine Neuheit dar. Der folgende Abschnitt beschreibt das Konzept.

Analyseberichte

Visual Studio besitzt, wie auch andere IDEs, ein Fenster zum Anzeigen von Fehlern im Quellcode (s. Abbildung 4.4). Roslyn unterstützt das Melden von Codeanalysen in diesem Fenster. Verantwortlich dafür ist die Klasse `DiagnosticDescriptor`. Der Eintrag in dem Fehlerfenster bezieht seine Daten aus diesem Descriptor, darunter auch die Analyse-ID, eine Textbeschreibung und die Warnstufe. Ein Nachteil ist jedoch, dass diese Klasse als `sealed` markiert wurde und daher nicht als Basisklasse dienen kann. Daraus folgt, dass weitere wichtige Informationen, wie sie die Anforderung **F1.3** beschreibt, dem Descriptor nicht hinzugefügt werden können. Um jedoch trotzdem detaillierte Informationen zu einer Codeanalyse anbieten zu können, musste ein erweitertes Konzept erarbeitet werden. Die Idee ist einen Analysebericht zu entwerfen, der dem `DiagnosticDescriptor` ähnlich ist, sodass einige Informationen wie ID und Beschreibung übernommen werden können, aber auch Krypto-API spezifische Angaben zulässt. Die Schnittstelle zu diesem Konzept heißt `IAnalysisReport` und wird in Listing 4.6 wiedergegeben. Identisch wie bei dem Descriptor, muss ein Analyzer einen oder mehrere dieser Berichte angeben, die der Analyzer zur Verfügung stellt. Möchte der Analyzer nun eine Codeanalyse melden, so muss er dies dennoch über einen Descriptior machen, da nur so die Interaktion mit dem Fehlerfenster von Visual Studio erfolgen kann. Um nun dennoch den hier vorgestellten Analysebericht aufzurufen, wurde folgendes System eingeführt:

Meldet ein Roslyn Analyzer eine Analyse, so gibt Visual Studio dem Entwickler die Möglichkeit durch das Klicken auf den Fehlercode (s. Abbildung 4.4, erste Spalte) eine Aktion durchzuführen. Standardmäßig führt diese Aktion eine Internetsuche dieses Fehlercodes durch, jedoch lässt sich diese überschreiben. In Sharper Crypto-API Analysis wird stattdessen die Verwaltungsinstanz befragt, ob ein Analysebericht mit diesem Fehlercode von einem Analyzer verfügbar ist. Falls dies zutrifft, wird das Toolfenster von Sharper Crypto-API Analysis geöffnet und der Bericht wird angezeigt (s. Abbildung 4.5). Eine Auflistung aller verfügbaren Analyseberichte (Anforderung **F1.4**) erfolgt durch Klicken auf den dritten Buttons in der oberen Leiste des Toolfensters.

[36]Die Codeanalyse wird durch das Aufrufen der Methode `Initialize` der `DiagnosticAnalyzer` Klasse. Wann diese Methode aufgerufen wird wird von Visual Studio entschieden.

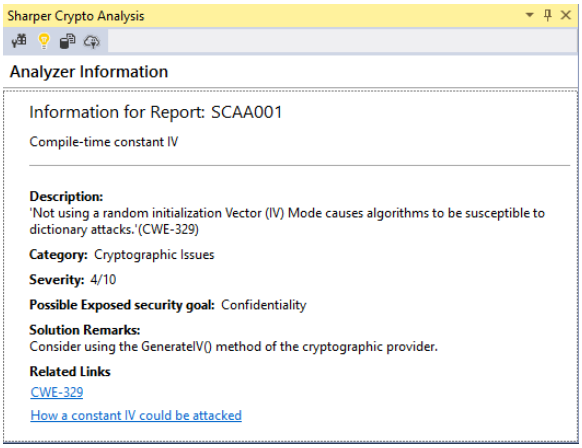

Abbildung 4.5: Analysebericht in Sharper Crypto-API Analysis

```
1  public interface IAnalysisReport
2  {
3  /// <summary>
4  /// Contains custom data to be displayed in a reports viewer
5  /// <remarks>Object should be a UI component like
       FrameworkElement</remarks>
6  /// </summary>
7  object AdditionalContent { get; }
8  string Category { get; }
9  string Description { get; }
10 string Id { get; }
11 Uri MoreDetailsUrl { get; }
12 IEnumerable<NamedLink> RelatedLinks { get; }
13 string Severity { get; }
14 string SolutionRemarks { get; }
15 string Summary { get; }
16 /// <summary>
17 /// Security goals like Authenticity or Integrity which could
       get exposed
18 /// </summary>
19 SecurityGoals ExposedSecurityGoals { get; }
20 }
```

Listing 4.6: Schnittstelle eines Codeanalyseberichts

4.5.4 Codegenerierung

Neben den Analyzern spielt die Codegenerierung von kryptographischen Aufgaben eine sehr wichtige Rolle in Sharper Crypto-API Analysis. CogniCrypt wird hier als Vorbild herangezogen. So wird das Erstellen der Krypto-Aufgabe auch hier durch einen Nutzerassistenten begleitet. Dieser Abschnitt beschreibt Funktionsweise und Implementation der Komponenten, die diese Funktionalität schaffen.

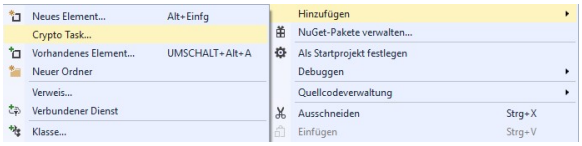

Abbildung 4.6: Hinzufügen einer Krypto-Aufgabe in Visual Studio

Verfügbare Aufgaben

Bevor jedoch auf die technischen Aspekte Codegenerierung eingegangen wird, gibt dieser Abschnitt einen kurzen Überblick, welche kryptographischen Codegenerierungsaufgaben im aktuellen Entwicklungsstand vorhanden sind und wie der Benutzer von Sharper Crypto-API Analysis auf die Codegenerierung zugreifen kann.

Wie in Abschnitt 4.4.2 erläutert, konzentriert sich diese SCAT zunächst einmal auf Probleme bei der Anwendung von symmetrischen Verschlüsselungsverfahren. Die Codegenerierung ist ein Werkzeug, welches es ermöglicht ganze Klassen zu erzeugen, ohne dass der Programmierer auch nur eine Zeile schreiben muss. Da sich Entwickler schwer tun, die richtigen Methoden einer Krypto-API auszuwählen oder das Funktionsangebot der API nicht in ihren Anwendungskontext unterbringen können, empfiehlt sich die Generierung einer Wrapper-Klasse, die durch selbsterklärende Methoden wie `SimpleEncryptWithPassword(string secretMessage, string password)` einfacher für den Programmierer zu benutzen sind. Die Generierung eines solchen Wrappers ist daher eine betrachtete Aufgabe. Um nicht nur eine, sondern mindestens zwei verschiedene Krypto-Aufgabe zu unterstützen, wird auch ein Wrapper zur Generierung von zufälligen Schlüsseln als Krypto-Aufgabe durch die Codegenerierung angeboten.

Visual Studio bietet mehrere Möglichkeiten neue Elemente wie Dateien und Ressourcen zu einem Softwareprojekt hinzuzufügen. Eine dieser Möglichkeiten besteht durch das Aufrufen des Kontextmenüs im Projektmappen-Explorer. Dieses Kontextmenü besitzt auch den Eintrag „Hinzufügen“. In diesem Untermenü wurde der Menüeintrag untergebracht, der die Codegenerierung von Sharper Crypto-API Analysis ausführt. Abbildung 4.6 zeigt diese Menüführung. Die Platzierung in dieses Menü wurde gewählt, um dem Nutzer zu verdeutlichen, dass nach der Codegenerierung der Krypto-Aufgabe eine neue Datei in das Softwareprojekt eingefügt wird. Durch das Klicken auf den Menüpunkt *Crypto Task...* wird der Nutzerassistent aufgerufen.

Nutzerassistent

Der Nutzerassistent, auch Wizard genannt, hilft dem Programmierer bei der Auswahl und Konfiguration einer Krypto-Aufgabe. Gemäß der Anforderung **NF3.1** wurde darauf geachtet, dass der Assistent eine für den Entwickler möglichst natürliche Sprache verwendet. Die Abbildung 4.7 zeigt das Auswahlfenster des Assistenten für Krypto-Aufgaben. Wie zu erkennen ist, behandelt die ausgewählte Aufgabe die symmetrische Verschlüsselung. Dieser Begriff wird jedoch bewusst nicht verwendet, um auch Entwickler anzusprechen, die diesen Begriff nicht kennen oder verstehen. Stattdessen bemüht sich die Beschreibung das kryptographische Verfahren aufgabenorientiert zu erklären und richtet sich persönlich an den Programmierer. Die Benutzeroberfläche orientiert sich stark an den in Windows bekannten Installationsassistenten (s. Anfor-

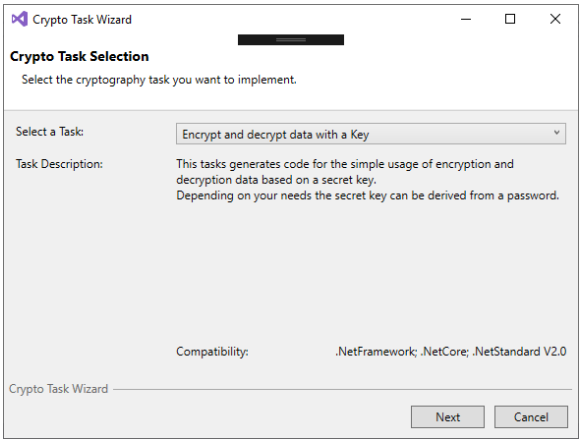

Abbildung 4.7: Benutzerassistent für Krypto-Aufgaben in Sharper Crypto-API Analysis

derung **NF3.0**). Je nach ausgewählter Aufgabe stehen dem Nutzer noch verschiedene Einstellungsmöglichkeiten zur Verfügung, mit welchen er den zu generierenden Code an seine Bedürfnisse anpassen kann. Im Beispiel der symmetrischen Verschlüsselung kann er unter anderen angeben, ob die Verschlüsselung ein Passwort vorsieht oder wie sicher das benutzte Verfahren sein soll (s. Abbildung 4.8). Auch bei den Einstellungsmöglichkeiten wurde darauf geachtet eine möglichst einfache Sprache zu benutzten.

Die Implementierung dieses Assistenten erfolgt seitenbasiert nach dem MVVM Pattern[37]. Das heißt, dass jede einzelne Seite eine eigene Komponente für sich darstellt. In der C# Programmierung hat sich das MVVM Pattern, eine Variante des MVC Pattern, bewährt, um die Anwendungslogik und Darstellung der UI-Elemente zu trennen. Pattern (dt. Entwurfsmuster) sind „bewährte [...] Lösungen für Probleme, die immer wieder [...] auftreten“ (Siebler, 2014, S. 1). Das ViewModel, siehe Listing 4.7, einer Seite enthält eine Referenz auf die jeweils nächste und vorherige Seite. Mit der Eigenschaft `CanFinish` wird festgelegt, ob diese Seite den Assistenten beenden kann. Die Eigenschaft `DataModel` enthält im Anwendungsfall der Codegenerierung das Datenmodell der ausgewählten Aufgabe, die generiert werden soll.

```
1  public interface IWizardPage : INotifyPropertyChanged
2  {
3      string Name { get; }
4      string Description { get; }
5      FrameworkElement View { get; }
6      object DataModel { get; set; }
7      IWizardPage NextPage { get; set; }
8      IWizardPage PreviousPage { get; set; }
9      bool CanFinish { get; }
10 }
```

Listing 4.7: Vereinfachte Schnittstelle einer Seite des Assistenten

[37] https://blogs.msdn.microsoft.com/msgulfcommunity/2013/03/13/understanding-the-basics-of-mvvm-design-pattern/, Zugriff: 15.12.2018

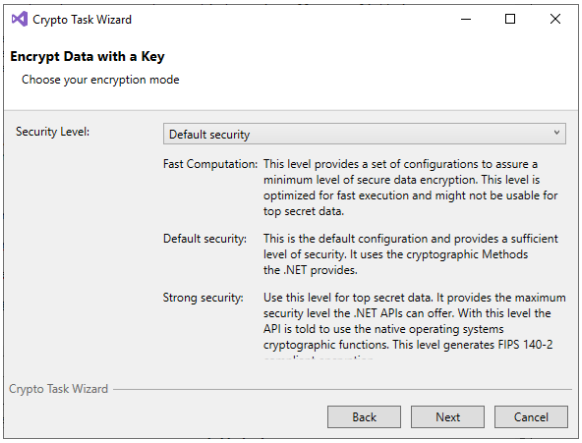

Abbildung 4.8: Benutzerassistent für eine symmetrische Verschlüsselung

Implementierung einer Krypto-Aufgabe

Will ein Erweiterungsentwickler eine neue Krypto-Aufgabe dem SCAT hinzufügen, so muss er eine Klasse schreiben, die die Schnittstelle `ICryptoCodeGenerationTask` (s. Listing 4.8) implementiert. Die Eigenschaften `Name` und `Description` werden für das Auswahlfenster des Nutzerassistenten benötigt. Falls die Krypto-Aufgabe Einstellungsmöglichkeiten bietet, können diese zusammen mit dem Dateinamen der zu generierenden Datei im Modell angegeben werden. Die Eigenschaft `WizardProvider` beinhaltet eine Klasse, die alle verfügbaren Assistentseiten sammelt und dem Nutzerassistenten zur Verfügung stellt. Der `TaskHandler` ist für die Codegenerierung anhand des Modells zuständig. Mehr zur Generierung im nächsten Unterabschnitt.

Zum Abschluss muss der Erweiterungsentwickler seine Klasse mit der Codezeile `[Export(typeof(ICryptoCodeGenerationTask))]` registrieren. Abschnitt 4.5.6 erklärt diesen Schritt.

```
1 public interface ICryptoCodeGenerationTask
2 {
3     string Name { get; }
4     string Description { get; }
5     ICryptoCodeGenerationTaskModel Model { get; }
6     ICryptoCodeGenerationTaskHandler TaskHandler { get; }
7     ICryptoTaskWizardProvider WizardProvider { get; }
8 }
```

Listing 4.8: Vereinfachte Schnittstelle einer Krypto-Aufgabe

Template Engine

Das SDK von Sharper Crypto-API Analysis bietet eine abstrakte Klasse namens `CryptoCodeGenerationTaskHandler` an, mit der anhand eines Datenmodells Codedateien generiert werden können. Zentrales Hilfswerkzeug ist eine Klasse, die es ermöglicht, Platzhalter in einem `string`, als dem Inhalt einer Quellcodedatei, mit Werten auf einem Datenmodell zu substituieren.

Die in dieser Arbeit implementierte Engine unterstützt einzeilige Platzhalter mit dem Format: `$PlatzhalterName$`. Eine Verschachtelung mehrerer Platzhalter wie `$a$b$$`, wonach zuerst `b` und danach `a` substituiert wird, ist nicht vorgesehen. Würde man also `$a$b$$` mit `a = x` und `b = y` ersetzten, wäre das Resultat dieser Engine `xb$$`.

Im aktuellen Entwicklungsstand können dank dieser Engine beispielsweise verschiedene .NET Implementationen der AES-Verschlüsselung, je nach Sicherheitsstufe, ausgetauscht werden. Die Template Engine ist im SDK des SCAT verfügbar.

4.5.5 Konfiguration und Erweiterungsmanagement

Das Erweiterungsmanagement von Sharper Crypto-API Analysis schließt sowohl die Anforderungen **F4** (Konfigurierbarkeit), als auch **F5** (Erweiterbarkeit durch den Benutzer) mit ein. Dieser Abschnitt beschreibt zunächst welche Konfigurationsmöglichkeiten in dem SCAT verfügbar sind, und wie der Benutzer diese vornehmen kann. Es wird ein Git-basiertes Konfigurationsverwaltungssystem vorgestellt und anschließend beschrieben. Der dritte Teil dieses Abschnittes geht auf die Verwaltung von hinzufügbaren Programmerweiterungen, den hier genannten Extensions, ein.

Konfiguration der Analyzer

Welche Einflüsse False Positives oder False Negatives auf die Benutzbarkeit eines SCAT haben, wurde in dieser Arbeit bereits erläutert. Je nach Anforderungen der zu entwickelnden Software oder dem Arbeitsflusses des Programmierers, der dieses SCAT verwendet, können bzw. müssen Codeanalysen anders eingeschätzt werden[38]. Visual Studio bietet zwar die Möglichkeit, Analyzer pro Codezeile oder Datei abzuschalten, jedoch nicht für das gesamte Softwareprojekt. Daher können in den Einstellungen von Sharper Crypto-API Analysis, Analyzer gänzlich abgeschaltet werden (s. Abbildung 4.9).

Des Weiteren erlaubt das SCAT dem Benutzer das Einstellen einer Analysehärte, die durch die Enumeration `AnalysisSeverity` implementiert ist. Die Änderung dieser Einstellung wird automatisch auf alle installierten Codeanalysen für Sharper Crypto-API Analysis angewendet. Entwickler von Codeanalysen können je nach eingestellter Analysehärte eine andere Meldung oder Warnstufe angeben. So empfiehlt sich beispielsweise bei der Einstellung `Low` die Codeanalyse mit der Warnstufe Mitteilung zu melden oder die Meldung ganz zu unterdrücken. In wie weit der Codeanalyseentwickler die Analysehärte interpretiert, ist jedoch ihm überlassen.

Die hier beschriebenen Einstellungen werden in der IDE gespeichert, sodass diese Konfiguration auch nach einem Neustart der IDE wieder zur Verfügung steht.

Konfigurationsverwaltung

IDEs, darunter auch Visual Studio, erlauben das Speichern von Benutzereinstellungen der IDE auf einem Server. Um diese Funktionalität zu nutzen, müssen sich die

[38] Der Fall das Codeanalysen vernachlässigt werden müssen, kann beispielsweise durch die Entwicklung einer Software begründet werden, die ein älteres System unterstützen muss. Diese ältere System könnte als Verschlüsselungsverfahren DES benutzen. Die neue Software muss demnach auch dieses Verfahren verwenden. Das SCAT würde jedoch feststellen, dass ein unsicheres Vefahren gewählt wurde.

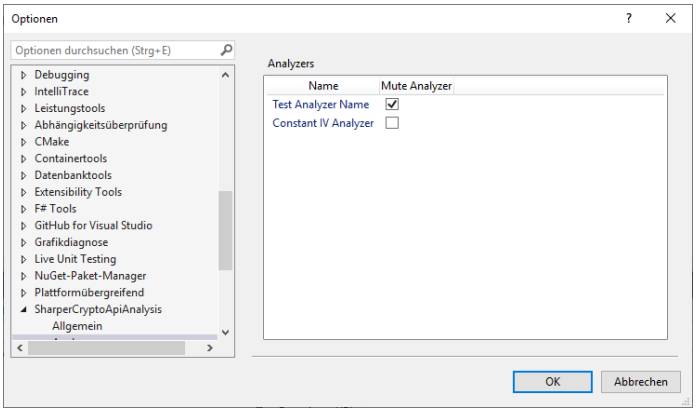

Abbildung 4.9: Einstellungsbildschirm zum globalen Abstellen von Codeanalysen

Entwickler mit einem Microsoft-Account in der IDE anmelden. Startet er nun Visual Studio auf einem anderen Rechner und logt sich ebenfalls mit seinem Account ein, synchronisiert Visual Studio die Benutzereinstellungen automatisch. Zu diesen Einstellungen gehören unter anderem das Farbthema oder Tastaturbelegungen. Hingegen werden Einstellungen von Plugins nicht übernommen, sodass das manuelle Konfigurieren der IDE, darunter auch das Installieren aller Plugins neu durchgeführt werden muss. Hier besteht die Unannehmlichkeit, dass der Programmierer vielleicht seine ganzen Plugins und Einstellungen nicht auswendig kennt. Eine weitere Schwierigkeit, die dieses Szenario beinhaltet, wird deutlich, wenn anstatt nur einem Programmierer ein ganzes Team betrachtet wird. Auch wenn es möglich wäre, Einstellungen von Plugins mit einem Account zu verbinden, so müsste sich jeder im Team, der die gleichen Einstellungen verwenden will, mit dem selben Account einloggen. Sofern es sich um ein Privatkonto handelt, müssten die Anmeldedaten dieses Accounts an die übrigen Entwickler des Teams preisgegeben werden oder der Besitzer des Kontos richtet die IDE aller Teammitglieder ein. In beiden Fällen besteht die Bedrohung, dass entweder mit den Anmeldedaten oder mit dem Account selbst Missbrauch betrieben wird.

Um die Probleme dieses Szenarios zu lösen, wurde in Sharper Crypto-API Analysis ein System entwickelt, um Plugin-spezifische Einstellungen mit Teammitgliedern oder anderen IDE Instanzen auszutauschen. Mittels Versionskontrollsystemen wie Git, verwalten Softwareentwickler ihren Code oder tauschen ihn mit anderen aus. Jeder Entwickler hat seinen eigenen Account auf der Git-Plattform und kann trotzdem auf die Dateien anderer zugreifen. Die Idee, dass Quelltextdateien in einem Repository dieser Plattformen abgelegt werden können, lässt sich natürlich auch auf andere Dateien projizieren, darunter auch Konfigurationsdateien. Die Voraussetzung, dass eine Konfiguration in einem Git Repository abgelegt werden kann, ist lediglich, dass das Plugin auf diese Konfiguration zugreifen und sie verarbeiten kann. Für diese Arbeit wurde sich daher für ein XML-Markup entschieden, welches im Abschnitt 4.5.7 genauer beschreiben wird.

Weitere Alternativen zum Austausch von Konfigurationen, darunter auch ein einfacher Dateiserver oder ein REST-basierter Webservice, können ebenfalls in Betracht gezogen werden. Die besondere Eigenschaft bei der Benutzung eines Git Repository ist jedoch, dass die Konfigurationsdaten des SCAT und der Quelltext der Anwen-

dung, der durch das SCAT analysiert werden soll, die gleiche Quelle, nämlich das Repository, haben. Dadurch versprechen sich die folgenden Vorteile:

- Jedes Repository kann seine eigene Konfiguration bereitstellen, die auf die Anforderungen der darin liegenden Software perfekt angepasst sind.
- Die Entwickler, die eine Konfiguration benutzen wollen, müssen sich nicht weitere Informationen merken, als die Repositoryadresse der Software, die sie programmieren wollen.
- Da die Konfigurationsdatei selbst ein XML-Markup ist, kann sie durch Git entsprechend versioniert und verwaltet werden.

Ein Repository, welches eine Konfiguration für Sharper Crypto-API bereitstellt, wird im weiteren Verlauf auch als Konfigurationsrepository bezeichnet.

Im aktuellen Entwicklungsstand lassen sich durch die Konfigurationsdatei in einem Git Repository einerseits die oben beschrieben Analysehärte sowie die zu installierenden Extensions angeben. Die Dateien der Extensions können ebenfalls auf dem Git Repository abgelegt werden. Wie ein solches Repository aufzubauen ist, erklärt der Abschnitt 4.5.7.

Sharper Crypto-API unterstützt derzeit zwei Synchronisationsmodi. Mit einem Synchronisationsmodus wird angegeben, wie sich das SCAT bei verfügbaren Änderungen der Konfiguration im Konfigurationsrepository verhalten soll. Der Modus *Soft* besagt, dass der Entwickler seine IDE frei konfigurieren kann. Alle Konfigurationen aus dem Konfigurationsrepository können lokal überschrieben werden und dienen lediglich als Empfehlung bzw. Startkonfiguration. Der Modus *Hard* verbietet diese Freiheiten. Alle installierten Plugins mit diesem Modus, verfügen somit über die selben Einstellungen.

Zur Nutzung eines Konfigurationsrepository in Sharper Crypto-API Analysis, muss der Benutzer im Toolfenster des SCAT unter dem Punkt „Manage Connections“ die URL des Repository angeben und auf den Button „Connect“ klicken. Die Abbildung 4.10 zeigt diese Benutzeroberfläche. Die Statusindikatoren geben an, ob die Verbindung hergestellt wurde und welcher Synchronisationsmodus verwendet wird. In zukünftigen Updates des SCAT ist jedoch vorgesehen, dass es automatisch eine verfügbare Konfiguration anwendet, sofern die IDE mit einem Git Repository verbunden ist. Die hier gezeigte Benutzeroberfläche wird daher obsolet. Ebenfalls unterstützt das Plugin derzeit nur GitHub als Anbieter eines Konfigurationsrepository.

Paketverwaltung

Das Erweiterungsmanagementsystem und insbesondere die in Sharper Crypto-API Analysis verwendete Benutzeroberfläche, zu sehen in Abbildung 4.11, orientiert sich ganz stark an dem Paketmanager *Nuget* von Microsoft. Der hier verwendete Code der Benutzeroberfläche wurde an vielen Stellen in vereinfachter Form aus dem Quellcode des Visual Studio Plugin für NuGet übernommen. Dieser Quellcode ist auf GitHub frei verfügbar[39]. Diese Übernahme kann durch die Anforderung **NF3.0**

[39]https://github.com/NuGet/NuGet.Client, Zugriff: 18.12.2018

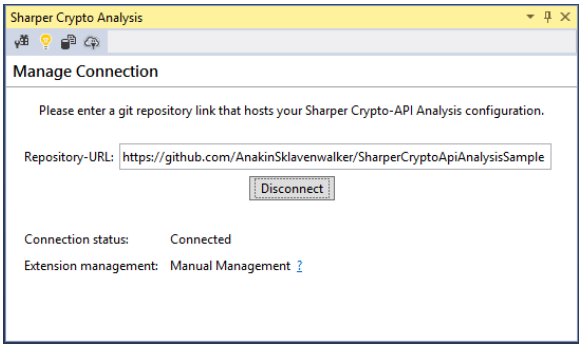

Abbildung 4.10: Toolfenster in Sharper Crypto-API Analysis zur Verbindung mit einem Konfigurationsrepository

begründet werden. NuGet ist in der C#-Entwicklung fest etabliert und viele Softwarebibliotheken werden so vertrieben. Der Benutzer kann sein Wissen über die Bedienung von NuGet direkt auf Sharper Crypto-API Analysis anwenden. Realisiert wird das Paketmanagement in diesem SCAT durch eine Implementation der Schnittstelle `IExtensionManager`. Sie stellt unter anderem Methoden bereit, um nach Updates zu suchen oder Extensions zu installieren, löschen oder zu updaten.

Extensions können neben Assemblies, die Codeanalysen und Codegenerierungen beinhalten, noch weitere von Visual Studio unterstützte Erweiterungstypen sein[40]. Um also eine Abstraktion zwischen der Extension und der Datei, die diese Extension enthält zu schaffen, wurde ein Metadatenkonzept entworfen. Die Idee ist, dass alle relevanten Informationen einer Extension über eine Schnittstelle namens `ISharperCryptoApiExtensionMetadata` verfügbar sind. Zu diesen Informationen gehören der Name der Extension, eine Kurzbeschreibung, die aktuelle Version, der Dateityp und ein Link, unter welchem die Extension verfügbar ist. Ein Cachedatei namens `AvailableExtensions.metadata` am Installationspfad aller Extensions sorgt dafür, dass die Paketverwaltung des SCAT diese Metadaten nicht jedes Mal vom Konfigurationsrepository herunterladen oder anhand der installierten Dateien generieren muss. Dadurch werden Netzzugriffe und aufwendige IO-Operationen eingespart.

Im aktuellen Entwicklungsstand wird das Paketmanagementsystem jedoch nur unterstützt, falls der Benutzer ein Konfigurationsrepository angegeben hat.

4.5.6 Erweiterbarkeit

Während sich der vorherige Abschnitt unter anderem mit dem Erweiterungsmanagement bei der Verwendung von Sharper Crypto-API Analysis auseinander gesetzt hat, widmet sich dieser Abschnitt den Themen, durch welche Technologie die Erweiterbarkeit des Tools umgesetzt wird und welche Besonderheiten beim Umgang mit dem SDK zu beachten sind.

[40]Eines dieser weiteren Typen sind sogenannte ItemTemplates, die ähnlich wie die hier vorgestellte Codegenrierung funktionieren und neue Dateien im Softwareprojekt erzeugen. Diese ItemTemplates werden jedoch nicht als Assembly, sondern im Dateiformat `.zip` bereitgestellt.

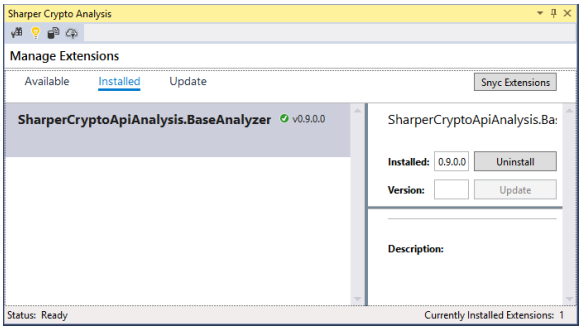

Abbildung 4.11: Benutzeroberfläche des Extensionmanagements in Sharper Crypto-API Analysis.

Managed Extensibility Framework (MEF)

Die Erweiterbarkeit von Anwendungen durch Plugins spielt auch in Sharper Crypto-API Analysis eine bedeutende Rolle. Entwickler sollen in der Lage sein, neue Extensions für das SCAT zu entwickeln, die dann verwendet werden können. Üblicherweise werden in C# Assemblies durch eine Referenz im Softwareprojekt integriert. Durch diese Referenz kann die Anwendung dann auf die Assembly zugreifen. Dieses Vorgehen setzt allerdings voraus, dass die zu verwendende Assembly zum Zeitpunkt des kompilierens bekannt ist. Das Problem, auf das man stößt, ist, dass bei der Entwicklung dieses SCAT nicht bekannt sein kann, welche Erweiterungen in Zukunft entwickelt werden. Das Referenzverfahren ist daher nicht brauchbar. Abhilfe schafft hier das MEF. Voraussetzung ist, dass die Anwendung, die erweitert werden soll, Verträge, beispielsweise in Form von Schnittstellen, bereitstellt, die der neuen Erweiterung bekannt sind. Hierzu ein Beispiel:

Ein SCAT soll durch das Hinzufügen von Assemblies neue Codeanalysen unterstützen. Das SCAT stellt in seinem SDK dazu die Schnittstelle `IAnalyzer` (s. Listing 4.9) zur Verfügung. Der Erweiterungsentwickler muss nun für eine neue Codeanalyse eine Klasse erstellen, die diese Schnittstelle implementiert (s. Listing 4.10). Er muss sich nicht darum kümmern, mit welchen Parametern und wann die in der Schnittstelle spezifizierte Methode `Analyse(string codeToAnalyse)` aufgerufen wird, da diese Aufgabe das SCAT übernimmt. Es müssen nur noch die folgenden zwei Dinge beachtet werden:

1. Wie in der vierten Zeile in Listing 4.10, muss der Erweiterungsentwickler die Klasse `SampleAnalyzer` unter dem Namen der Schnittstelle `IAnalyzer` beim MEF registrieren. Dazu ist der Verwendung des `ExportAttribute` erforderlich. Verlangt das SCAT nun vom MEF eine oder mehrere registrierte Klassen dieser Schnittstelle, gibt das MEF dem SCAT die gewünschten Instanzen. Listing 4.11 zeigt eine Implementierung einer Verwalterklasse, die sich um die Ausführung der verfügbaren Analyzer kümmert. Entscheidend ist die sechste Zeile. Durch das `ImportManyAttribute` symbolisiert das SCAT dem MEF, dass es alle verfügbaren Instanzen der Schnittstelle `IAnalyzer` haben will.

2. Der Erweiterungsentwickler muss seine Assembly an einer Stelle installieren, die von dem SCAT als geeigneter Ort für Erweiterungen festgelegt wurde. Das MEF sucht beim Start der Anwendung an diesem Ort nach verfügbaren Assemblies

und lädt sie in die CLR, sodass sie benutzt werden können. Üblicherweise ist ein geeigneter Installationsort der Pfad, an dem sich auch die ausführbare Datei der Anwendung befindet.

```
1  // Assembly: SCAT
2  public interface IAnalyzer
3  {
4      IAnalysisResult Analyse(string codeToAnalyse);
5  }
```

Listing 4.9: Beispielschnittstelle IAnalyzer zur Demonstration von MEF

```
1  // Assembly: Extension
2  using System.ComponentModel.Composition;
3
4  [ExportAttribute(typeof(IAnalyzer))]
5  public class SampleAnalyzer : IAnalyzer
6  {
7     public IAnalysisResult Analyse(string codeToAnalyse)
8     {
9         // Logic here...
10    }
11 }
```

Listing 4.10: Beispielklasse SampleAnalyzer zur Demonstration von MEF

```
1  // Assembly: SCAT
2  using System.ComponentModel.Composition;
3
4  internal class AnalyzerManager
5  {
6      [ImportManyAttribute] private IAnalyzer[] analyzers;
7
8      internal void AnalyseInternal(string codeToAnalyse)
9      {
10         foreach (var analyzer in analyzers)
11         {
12             var result = analyzer.Analyse(codeToAnalyse);
13             // Work with Result...
14         }
15     }
16 }
```

Listing 4.11: Beispielklasse AnalyzerManager zur Demonstration von MEF

Nach einem ähnlichen Prinzip, wie hier beschreiben, arbeiten auch Visual Studio und Roslyn. Sharper Crypto-API Analysis nutzt das MEF, um sich selbst bei Visual Studio als Plugin zu registrieren, um Verwaltungsdienste unter Schnittstellen wie `IExtensionManager` zur Verfügung zu stellen oder um Codegenerierungsaufgaben

beim Generierungsassistenten anzumelden. Ebenfalls können Extensions für Sharper Crypto-API Analysis als MEF Komponente registriert werden. Hier liegt auch der Grund, warum bei der Installation neuer Extensions die IDE neugestartet werden muss. Sowohl MEF als auch Visual Studio bieten keine Funktion, neue Assemblies zur Laufzeit neu einzubinden.

Als Zusatz soll hier nicht unerwähnt bleiben, welche anderen Faktoren beim Installieren neuer Extensions erforderlich sind, damit diese von Visual Studio ordnungsgemäß geladen werden. Jedes Plugin für Visual Studio besitzt an seinem Installationsort eine Xml-Datei namens `extension.vsixmanifest`. Dieses Markup definiert unter anderem, welche Assemblies MEF-Komponenten oder Analyzer beinhalten. Beim Start verarbeitet Visual Studio diese Datei und initialisiert entsprechend das MEF. Der Installationspfad für Extensions von Sharper Crypto-API Analysis befindet sich im Installationsverzeichnis unter dem Unterordner `\Extensions\`. Bei der Installation bzw. Deinstallation einer Extension schreibt oder löscht der `ExtensionManager` einen Eintrag in die `extension.vsixmanifest` Datei. Zusätzlich müssen lokale Caches von Visual Studio gelöscht werden, damit die Änderungen eintreffen. Insbesondere das Löschen dieser Caches wird in den Dokumentationen des Visual Studio SDK nicht erwähnt.

SDK Nutzungshinweise

Welche Assemblies zum SDK gehören, wurde bereits im Abschnitt 4.5.2 erläutert. Um das SDK für Erweiterungsentwickler verständlich zu halten, wurden einerseits alle verfügbaren Typen (Klassen, Schnittstellen, Enumerationen, etc.) mit Dokumentationskommentaren versehen, andererseits ermöglicht die quelloffene Entwicklung von Sharper Crypto-API Analysis, dass sich der Entwickler am bereits vorhandenen Code ein Beispiel nehmen kann. Einige wichtige Hinweise zur problemfreien Entwicklung werden hier kurz aufgeführt.

- **Vermeidung von vielen Abhängigkeiten:** Das Roslyn SDK[41], das Visual Studio SDK[42] und diese SDK[43] bieten bereits eine hohe Anzahl an verfügbaren Komponenten. Aufgrund der oben beschrieben Art und Weise, wie Visual Studio Erweiterungen lädt, sollte sich auf diese SDKs beschränkt werden, um eine höchstmögliche Kompatibilität zu gewährleisten.

- **Vermeidung von lokalen Kopien:** Wird eine Extension entwickelt, müssen die benutzten SDKs als Referenz angegeben werden. Um mögliche Probleme mit dem Erweiterungsmanagementsystem von Sharper Crypto-API Analysis zu vermeiden, wird empfohlen die Option *Copy Local* in Visual Studio auf `false` zu setzen (s. Abbildung 4.12).

- **Zielplattform:** Extensions für Sharper Crypto-API Analysis können sowohl für die Plattform .NET Framework, als auch für das .NET Standard entwickelt werden. Das .NET Standard eignet sich besonders für Extensions, die ausschließlich

[41] Assemblies mit dem Namepsace `Microsoft.CodeAnalysis.*`
[42] Assemblies mit dem Namepsace `Microsoft.VisualStudio.*`
[43] Assemblies mit dem Namepsace `SharperCryptoApiAnalysis.*`

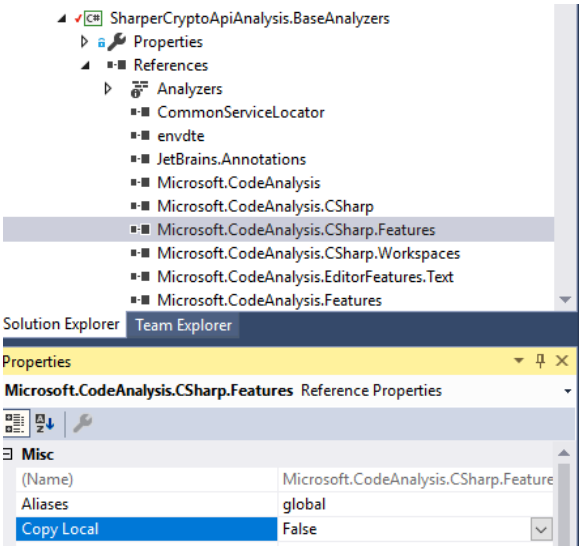

Abbildung 4.12: Einstellungsempfehlung bei der Entwicklung von Extensions

Roslyn-basierte Analyzer bereitstellen. Extenisons, die hingegen auch WPF Benutzeroberflächen, wie beispielsweise im Codegenerierungsassistenten benutzt werden, verwenden wollen, sollten sich als Zielplattform für .NET Framework entscheiden.

4.5.7 Gebrauchshinweise

Installation und Einsatz

Sharper Crypto-API Analysis ist ein freies und quelloffenes SCAT. Zur Installation und Verwendung wird Visual Studio 2017 vorausgesetzt. Auf der GitHub Seite des Tools kann die jeweils aktuelle Version heruntergeladen werden[44]. Die Installation erfolgt durch das Ausführen der Datei `SharperCryptoApiAnalysis.vsix`. Ein Installationsassistent erscheint und führt den Benutzer durch die Installation.

Um die Anforderung **NF1.3** zu erfüllen, wird beim nächsten Start der IDE ein Informationsfesnster, wie in Abbildung 4.13 zu erkennen, erscheinen. Daran und an dem neuen Menüeintrag in der Menüleiste kann man erkennen, dass das SCAT erfolgreich installiert wurde und einsatzbereit ist. Neben der Möglichkeit dieses Informationsfenster bei weiteren Starts der IDE zukünftig zu unterdrücken, hat der Nutzer auch die Möglichkeit, das Toolfenster von Sharper Crypto-API Analysis zu öffnen. Alternativ kann dieses auch über den Menüeintrag „Ansicht → Weitere Fenster“ erfolgen.

Um das aktuelle Funktionsangebot in ganzer Fülle nutzen zu können, sollte der Nutzer im Konfigurationsfenster des SCAT, wie im Abschnitt 4.5.5 beschrieben, das Repository seines Entwicklungsprojekts angeben und das Projekt in der IDE öffnen.

Möchte der Benutzer Sharper Crypto-API Analysis Versionshinweise des SCAT erhalten, es deaktivieren oder löschen, so kann er dies unter dem Menüpunkt „Extras → Extensions und Updates...“ erledigen (s. Abbildung 4.14).

[44]https://github.com/AnakinSklavenwalker/SharperCryptoApiAnalysis/releases, Zugriff: 14.12.2018

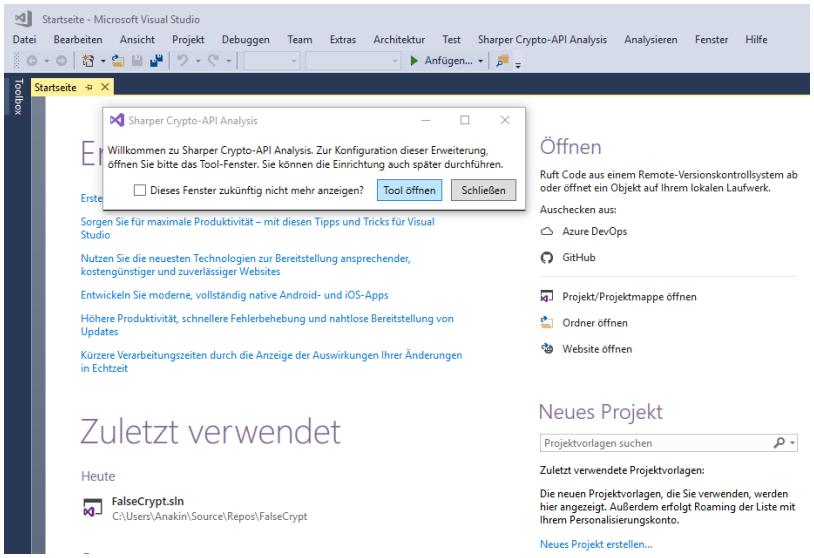

Abbildung 4.13: Installationshinweise von Sharper Crypto-API Analysis in Visual Studio

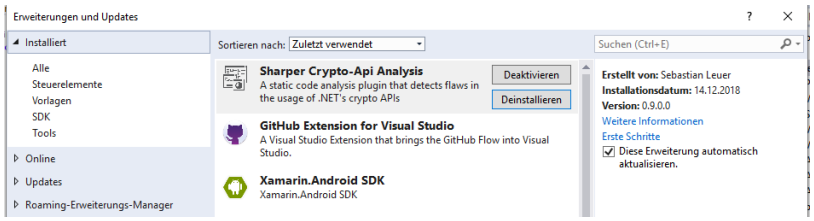

Abbildung 4.14: Erweiterungsmanager in Visual Studio

Repositoryeinrichtung zur Nutzung von Sharper Crypto-API Analysis

Bisher wurde im Abschnitt 4.5.5 nur beschrieben, wie man ein für Sharper Crypto-API Analysis eingerichtetes Git Repository als Konfigurationsquelle im Tool angeben kann. Dieser Abschnitt erklärt hingegen, welche Schritte notwendig sind, ein solches Git Repository einzurichten.

Grundsätzlich kann jedes Git Repository für die Konfiguration von Sharper Crypto-API Analysis herangezogen werden. Entscheidend ist, dass im Hauptverzeichnis (root directory) des Master-Branch die Datei `SharperCryptoApiAnalyzer.config` existiert. Diese wird vom SCAT beim Verbinden mit dem Repository heruntergeladen und verarbeitet. Der Inhalt der Datei ist ein XML-Markup, welches im aktuellen Entwicklungsstand mit dem Schema aus Listing 4.12 beschrieben werden kann. Die erforderlichen Angaben sind einerseits `<RepoAddress>`, welches die URL zum Repository enthält und andererseits `<SyncMode>`, der den Synchronisierungsmodus (s. Abschnitt 4.5.5) angibt. Eine Angabe der Analysehärte ist nicht erforderlich. Wird dieser Wert nicht angegeben, interpretiert das SCAT den Wert als `Default`. Ebenfalls wird in dieser Konfiguration festgehalten, welche Extensions für Sharper Crypto-API Analysis verfügbar sind und welche installiert werden sollen. Alle relevanten Metadaten der Extension werden hier angegeben. Das Element `<Type>` (Zeile 36) gibt an, um welchen Typ von Extension es sich handelt. Die Typen werden in der Enumeration `ExtensionType` spezifiziert. Enthält eine Extension sowohl einen Analyzer, als auch eine Codegenerierungsaufgabe, müssen beide Typen durch ein Komma getrennt angegeben werden. In diesem Beispiel würde das Element wie folgt aussehen: `<Type>MefComponent, Analyzer</Type>`. Die Reihenfolge der Typangaben ist egal.

```
1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="SharperCryptoAPIConfiguration">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element type="syncMode" name="SyncMode"/>
6          <xs:element type="xs:string" name="RepoAddress"/>
7          <xs:element type="severity"
               name="AnalysisSeverity" minOccurs="0"/>
8          <xs:element name="Extensions">
9            <xs:complexType>
10             <xs:sequence>
11               <xs:element name="Extension"
                     type="extension" maxOccurs="unbounded"
                     minOccurs="0"/>
12             </xs:sequence>
13           </xs:complexType>
14         </xs:element>
15       </xs:sequence>
16     </xs:complexType>
17   </xs:element>
18   <xs:simpleType name="syncMode" >
19     <xs:restriction base="xs:string">
```

```
20          <xs:enumeration value="Soft"/>
21          <xs:enumeration value="Hard"/>
22      </xs:restriction>
23    </xs:simpleType>
24    <xs:simpleType name="severity" >
25      <xs:restriction base="xs:string">
26          <xs:enumeration value="Default"/>
27          <xs:enumeration value="Strict"/>
28          <xs:enumeration value="Medium"/>
29          <xs:enumeration value="Low"/>
30          <xs:enumeration value="Informative"/>
31      </xs:restriction>
32    </xs:simpleType>
33    <xs:complexType name="extension">
34      <xs:sequence>
35          <xs:element type="xs:string" name="Name"/>
36          <xs:element type="xs:string" name="Type"/> <!--
              Comma separated list of enumeration:
              Interop.Extensibility.ExtensionType-->
37          <xs:element type="xs:string" name="Author"/>
38          <xs:element type="xs:string" name="Version"/>
39          <xs:element type="xs:string" name="Summary"/>
40          <xs:element type="xs:string" name="Description"/>
41          <xs:element type="xs:string" name="InstallPath"/>
42          <xs:element type="xs:boolean"
              name="InstallExtension"/>
43          <xs:element type="xs:boolean" name="External"/>
44          <xs:element type="xs:string" name="Source"/>
45      </xs:sequence>
46    </xs:complexType>
47  </xs:schema>
```

Listing 4.12: Schama der Konfigurationsdatei eines Git Repository zur Nutzung von Sharper Crypto-API Analysis

Zur einfacheren Umsetzung dieser Konfigurationsdatei in ein Git Repository, wurde neben dem SCAT auch zusätzlich eine Anwendung entwickelt, die einerseits die Konfigurationsdatei `SharperCryptoApiAnalyzer.config` generiert und automatisch einen Commit erstellt. Einen Push auf den Git Server unterstützt das Programm jedoch nicht. Die Abbildung 4.15 zeigt die Benutzeroberfläche der Anwendung. Der Nutzer gibt zunächst den lokalen Pfad des Repository ein, welches er für die Nutzung von Sharper Crypto-API Analysis konfigurieren möchte. In der Textbox „Git Storage Path“ wird der relative Pfad zum Hauptverzeichnes des Repositorys angegeben, in welchem die verfügbaren Extensions für das SCAT gespeichert werden sollen. Ein empfohlener Pfad ist beispielsweise `repoRoot\build\Analyzers`. Synchronisierungsmodus und Analysehärte können durch die Comboboxen angegeben werden. Die Auswahl von Extensions erfolgt durch Klicken des Buttons „Select Analyzers...“. Die Basiserweiterung `SharperCryptoApiAnalysis.BaseAnalyzers.dll` ist bereits in Sharper Crypto-API Analysis verfügbar und muss daher nicht angegeben werden.

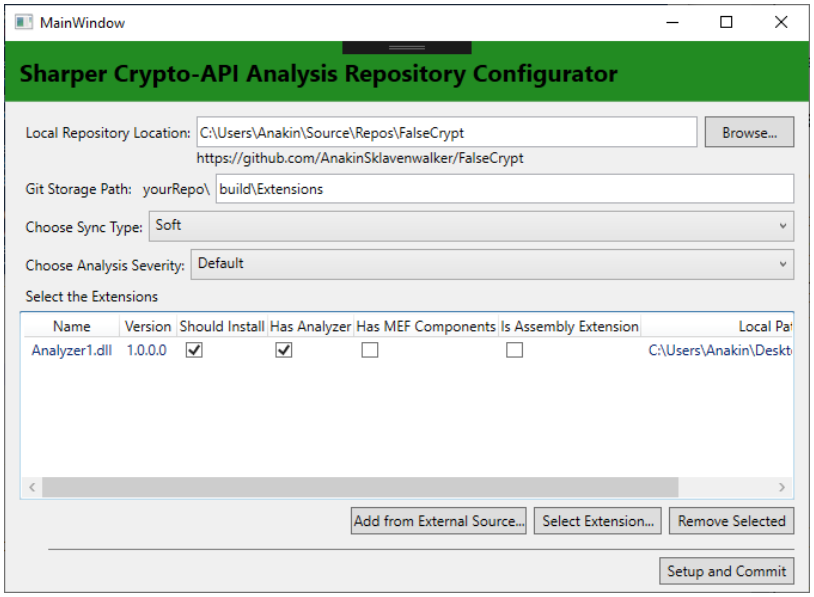

Abbildung 4.15: Git Repository Einrichtungsprogramm

Das Auswählen von externen Extensions, also Extensions, die auf einem fremden Dateiserver zur Verfügung stehen, wird in dieser Version noch nicht unterstützt. Hat der Benutzter die gewollten Extensions ausgewählt, werden mit einem Klick auf „Setup and Commit“ die ausgewählten Extensions an den vorher angegeben Pfad des Repository verschoben, die Konfigurationsdatei generiert und ein Commit erstellt, der dann gepusht werden kann. Zusätzlich wird eine weitere Konfigurationsdatei namens `SharperCryptoApiAnalyzer.private.config` erstellt, in welcher die lokalen Dateipfade der Extensions festgehalten werden. In künftigen Updates des Tools kann mittels dieser Information der Assistent die Liste der Extensions bei erneuter Verwendung wiederherstellen. Um Informationen über Systemkonfigurationen und Ordnerstrukturen zu schützten, wird diese zweite Datei automatische in die `.gitignore` Einstellung hinzugefügt und daher beim Commit nicht mit berücksichtigt.

Der Assistent ist ebenfalls im Git Repository von Sharper Crypto-API Analysis erhältlich.

Kapitel 5

Diskussion

In diesem Kapitel werden die gewonnenen Erkenntnisse aus dem vorherigem Abschnitt reflektiert betrachtet. Insbesondere wird dabei auf die eingangs gestellten Forschungsfragen aus dem Abschnitt 3.6 eingegangen. Diese werden hier noch einmal aufgeführt:
FF1: In welchen Punkten unterscheidet und gleicht sich die JCA von der .NET Krypto-API?
FF2: Können die in CogniCrypt erarbeiteten Java-Codeanalysen und Codegenerierungen auch für C# verwendet werden?
FF3: Können durch Konfiguration oder anderen Mechanismen Codemeldungen mit unterschiedlicher Einstufung durchgeführt werden?
FF4: Kann der Benutzer der IDE das Verhalten des SCAT ändern, um Analysen anders einzustufen, um somit auch eine bessere False Positive Rate oder False Negative Rate zu erlangen?
FF5: Welche Erweiterungs- und Konfigurationsmöglichkeiten sollte ein Roslyn-basiertes SCAT der Nutzung von Krypto-APIs für Visual Studio besitzen?
FF6: Ermöglicht die IDE Visual Studio das dynamische Laden bzw. Einfügen weiterer Analysekomponenten?

5.1 Unterschiede der .NET Krypto-API mit der Java Cryptography Architecture

Auf den ersten Blick erscheinen Java und C# sehr identisch. Beide Sprachen werden in Bytecode kompiliert, ihre Syntax ist an vielen Stellen gleich und ihre Frameworks (JDK und .NET Standard) ähneln sich im Funktionsumgang sehr. Die Unterschiede liegen im Detail, so auch bei den Krypto-APIs. Dieser Abschnitt will keine ausführliche Analyse liefern, welche Unterschiede genau zwischen der .NET Krypto-API und der JCA liegen, sondern anhand zweier Beispiele kurz aufzeigen, dass sich die APIs in ihrer Anwendung unterschiedlich handhaben.

In ihrem Funktionsumfang sind sich beide APIs sehr ähnlich. Symmetrische und asymmetrische Verschlüsselungen, Zufallsgeneratoren sowie Hashing- und Schlüsselableitungsfunktionen werden gleichermaßen unterstützt. An einigen Stellen bietet die JCA sogar mehr kryptographische Verfahren an. So können die Entwickler hier neben AES auch die RC-Derivate RC2, RC4 und RC5 sowie Blowfish verwenden. Die

Aspekte, in denen sich die APIs jedoch unterscheiden, liegen vor allem im Aufbau und der Handhabung.

Zunächst fällt auf, dass sich die Klassennamen beider APIs stark unterschieden. Werden in der .NET Krypto-API noch symmetrische Verschlüsselungsverfahren (Basisklasse: `SymmetricAlgorithm`) von asymmetrischen Verschlüsselungsverfahren (Basisklasse: `AsymmetricAlgorithm`) unterschieden, werden diese Verfahren in der JCA durch eine gemeinsame Klasse namens `Cipher` vereint. Hashfunktionen sind in .NET unter dem Namen `HashAlgorithm` und in Java unter `MessageDigest` erreichbar. Allein anhand dieser von den Java-Entwicklern gewählten Namensgebung kann verstanden werden, warum unerfahrene Programmierer Schwierigkeiten haben die JCA zu nutzen. Aus dem Namen `MessageDigest` geht auf den ersten Blick nicht unbedingt hervor, dass hier nicht nur die Namensverwandte MD5 Funktion verfügbar ist, sondern auch SHA-1 und SHA-2.

Ein weiterer großer Unterschied ist die Art und Weise, wie die APIs vom Programmierer verwendet werden müssen. Die JCA präferiert das sogenannte Facotry-Pattern, bei dem eine Klasseninstanz durch den Aufruf einer Methode zurückgegeben wird. Zur Bestimmung der kryptographischen Funktion, die instanziiert werden soll, muss ein eine Zeichenkette angegeben werden, welche die Konfiguration enthält[1]. Der Programmierer muss also wissen, welchen Wert er eingeben muss, um seine gewünschte Instanz zu erhalten. Eine direkte Instanziierung durch Aufruf des Konstruktors ist auf Grund seines Zugriffsmodifikators `protected` bzw. `private` nicht möglich. Aus einer Untersuchung geht hervor, dass der Einsatz des Factory-Pattern in APIs vermieden werden sollte (Ellis, Stylos & Myers, 2007). Die .NET Krypto-API besitzt zwar ebenfalls dieses Pattern, jedoch können die Klasseninstanzen auch über den Konstruktor der Klasse erzeugt werden.

> **FF1** wird daher wie folgt beantwortet: Der Funktionsumfang der JCA und der .NET Krypto-API unterscheidet sich nur geringfügig, auch wenn die JCA einige Verschlüsselungsverfahren mehr anbietet. Große Unterschiede liegen jedoch beim Aufbau der APIs vor. In der JCA werden Klasseninstanzen durch das Factory-Pattern erzeugt, in .NET Klassen durch Aufruf eines Konstruktors. Des Weiteren unterscheiden sich die Klassennamen beider APIs. Die JCA benutzt Klassennamen, die den Fachbegriff des implementierten Verfahrens beinhalten, während die .NET API vereinfachte Namen als Klassennamen benutzt. Zusammenfassend kann man die Meinung vertreten, dass die .NET Krypto-API gegenüber der JCA eine einfachere Handhabung besitzt.

5.2 Vergleich mit CogniCrypt

Für die Entwicklung von Sharper Crypto-API Analysis kann das statische Codeanalyseprogramm CogniCrypt als Vorbild betrachtet werden. Der wesentliche Unterschied zwischen beiden SCATs ist die Zielplattform. CogniCrypt untersucht Java Quelltext und ist derzeit für die IDE Eclipse erhältlich. Sharper Crypto-API Analysis konzentriert sich hingegen auf die Programmiersprache C# in Verbindung mit der Krypto-API aus dem .NET Standard 2.0. Das SCAT ist als Plugin für die IDE Visual Studio

[1] Ein Beispiel: `Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");`

verfügbar. Dieser Abschnitt geht auf weitere Gemeinsamkeiten und Unterschiede der beiden Tools ein.

5.2.1 Codeanalysen und Codegenerierung

Die Codeanalysen werden in den beiden SCATs durch verschiedene Technologien umgesetzt. CongiCrypt verwendet die Regelsprache CrySL. Sie arbeitet nach einem White-List Verfahren. Demnach wird sämtlicher Quelltext, der nicht mit den in CrySL spezifizierten Regeln konform ist, als Fehler gemeldet. Die Codeanalysen in Sharper Crypto-API Analysis basieren auf dem Roslyn SDK und verwenden ein Black-List Verfahren. Demnach werden in einem Analyzer Regeln festgelegt, die der Code nicht enthalten darf. Falls doch, meldet der Analyzer einen Analysebericht. Je nach verwendetem Verfahren, unterscheiden sich die False Positive und False Negative Raten. Ein White-List Verfahren, wie es CogniCrypt bietet, kann, sofern die Regel ein sicheres Verfahren abbildet, eine sehr gute False Negative Rate erreichen, da sämtliche Codeabweichungen von dieser sicheren Regel als Fehler gemeldet werden. Dem gegenüber bieten Black-List Verfahren eine bessere Toleranz für alternative Implementierungen, die unter Umständen genau so sicher sind, wie die Regel aus dem White-List Verfahren. Das SCAT kann dadurch bessere False Positive Raten erzeugen. Da zu hohe False Positive Raten schädlich für die Verwendung eines SCAT in der Programmierphase sind, wurde sich für Sharper Crypto-API Analysis für das Black-List Verfahren entschieden. Im Funktionsumfang der Codeanalysen ist CogniCrypt aufgrund der bereits länger fortgeschrittenen Entwicklungszeit Sharper Crypto-API Analysis voraus und unterstützt somit sowohl symmetrische, als auch asymmetrische Verschlüsselungsverfahren.

> **FF2** wird daher wie folgt beantwortet: Aufgrund der funktionalen Ähnlichkeiten zwischen der JCA und der .NET Krypto-API, können Codeanalysen wie Identifizierung schwacher Betriebsmodi sowohl in CogniCrypt, als auch in Sharper Crypto-API Analysis verwendet werden. Code kann aufgrund der verschiedenen Strukturen und Klassennamen der APIs sowie den unterschiedlichen Analysedienstleistern CrySL und Roslyn nicht wiederverwendet werden. Um die Nutzererfahrung mit dem SCAT durch eine geringe False Positive Rate zu verbessern, wurden die Analysen für Sharper Crypto-API Analysis nach einem Black-List Prinzip entwickelt.

Die Codegenerierung von Wrappern mit kryptographischen Verfahren, basierend auf aufgabenorientierten Angaben des Nutzers, ist eine Idee, die in Sharper Crypto-API Analysis von CogniCrypt übernommen wurde. Die Auswahl der zur Verfügung stehenden Codegenerierungen ist aufgrund des Zeitumfangs dieser Arbeit auf die Erzeugung zufälliger Schlüssel und einem Wrapper für eine symmetrische Verschlüsselung beschränkt. Die Implementierung der Codegenerierung und des Nutzerassistenten, durch welchen der Nutzer seine Angaben spezifiziert, wurde in dieser Arbeit neu entworfen.

5.2.2 Konfiguration und Erweiterbarkeit

In den Punkten Konfiguration und Erweiterbarkeit besteht bei CogniCrypt derzeit noch Entwicklungsbedarf. In der Version vom Dezember 2018 ist es bislang nicht

möglich, Einstellungen am SCAT und seinem Analyseverhalten vorzunehmen. Ein entsprechendes Ticket auf der Projektseite des Tools existiert bereits[2]. Ebenfalls ist der Benutzer von CogniCrypt nicht in der Lage, neue Codeanalysen, beispielsweise für andere Krypto-APIs, zu beziehen, außer sie werden direkt durch ein Softwareupdate des Plugins bereitgestellt. Mit CrySL können Erweiterungsentwickler oder die Hersteller der Krypto-APIs eigene Analyseregeln spezifizieren, diese jedoch nicht direkt dem Benutzer bereitstellen. Hier ist der Umweg über die Entwickler von CongniCrypt erforderlich, die das neue Regelwerk folglich als Update bereitstellen müssen. In Sharper Crypto-API Analysis war das Umsetzten dieser Funktionen bereits sehr früh geplant und wurde bei der Implementierung des Plguins entsprechend berücksichtigt.

5.2.3 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass sowohl CogniCrypt, also auch Sharper Crypto-API Analysis das Ziel haben, die Entwickler bei der Verwendung von Krypto-APIs zu unterstützen. Beide Tools sind in eine IDE integriert, sodass Fehler schon beim Schreiben des Quellcodes gefunden werden können. Ein erster Blick auf die verfügbaren Funktionen lässt die Vermutung zu, dass die beiden Tools, mit dem Unterschied, dass sie für verschiedene Programmiersprachen geeignet sind, recht ähnlich wirken. In ihrer Benutzung weisen sie jedoch einige Unterschiede auf. Die Codeanalysen von CogniCrypt decken die in Abschnitt 5.1 genannten fünf Funktionsbereiche der JCA ab. In Sharper Crypto-API Analysis wurde sich aufgrund des begrenzten Zeitrahmens nur auf einen Teil konzentriert. Als neue Funktion kann sich der Benutzer einen ausführlicheren Analysebericht anzeigen lassen, in welchem er noch weitere Informationen erhält. In den Punkten Konfiguration und Erweiterbarkeit ist Sharper Crypto-API Analysis CogniCrypt bereits voraus. Der Benutzer kann das Analyseverhalten selbst einstellen und nach Bedarf und Verfügbarkeit neue Erweiterungen für das SCAT hinzufügen.

5.3 Visual Studio als Entwicklungsplattform

Visual Studio ist eine sehr verbreitete und viel genutzte IDE. Für die .NET Programmiersprachen, darunter auch C#, stellt sie in gewisser Maßen ein Monopol dar. Weitere auf C# spezialisierte IDEs wie *MonoDevelop* oder *SharpDevelop* liegen mit weniger als einem Prozent Nutzungsanteil weit hinten[3]. Mit der Wahl von Visual Studio als Plattform für Sharper Crypto-API Analysis wird somit sichergestellt, dass das SCAT von den meisten C# Programmierern benutzt werden kann.

Das Visual Studio SDK ermöglicht es Erweiterungsentwicklern Plugins für die IDE zu entwickeln, die nicht nur sämtliche Funktionen neu hinzufügen können, sondern es sogar möglich machen, nahezu alle verfügbaren Funktionen zu erweitern oder auszutauschen. Obwohl dies auf den ersten Blick als riesiger Vorteil erscheint, konnten bei der Entwicklung von Sharper Crypto-API Analysis einige Schwierigkeiten festgestellt werden, die auch damit begründet werden können, dass der Autor dieser Arbeit

[2]https://github.com/eclipse-cognicrypt/CogniCrypt/issues/190, Zugriff: 20.12.2018
[3]http://pypl.github.io/IDE.html, Zugriff: 19.12.2018

bislang keine praktischen Erfahrungen mit dem SDK hatte. Als Einsteiger in die Erweiterungsentwicklung von Visual Studio ist man aufgrund des immensen Umfangs des SDK auf die Dokumentationen[4][5] und Codebeispiele angewiesen. Die verfügbaren Codebeispiele zeigen jedoch nicht die Details, die man bräuchte, um eine neue Funktionalität zu entwerfen. Als Beispiel kann hier die Überschreibung der Hilfsfunktion im Fehlerfenster der IDE herangezogen werden. In Abschnitt 4.5.3 wird diese Funktionalität beschrieben. Der Autor konnte keine Codebeispiele finden, die zeigen wie die Hilfsfunktion bzw. irgendeine Funktion im Fehlerfenster modifiziert werden kann. Auch waren dem Autor keine anderen Plugins mit dieser Funktion bekannt. Nur durch langes Durchsuchen der API und Dekompilierung von Assemblies der IDE[6] konnte eine Lösung gefunden werden.

Die folgende Beobachtung wird festgestellt: Visual Studio bietet dank eines umfangreichen SDK sehr viele Erweiterungsmöglichkeiten. Das Entwickeln von Plugins erfordert jedoch eine gewisse Erfahrung im Umgang und Angebot des SDK.

Die Anpassbarkeit und Erweiterbarkeit spielt in der Entwicklung von Sharper Crypto-API Analysis eine wichtige Rolle. So umfangreich die SDKs von Visual Studio und Roslyn auch sein mögen, lässt sich dennoch nicht garantieren, ob die gewünschten und erforderlichen Anforderungen zur Konfigurierbarkeit und Erweiterbarkeit überhaupt oder zumindest in einem Zeit- und Arbeitsaufwand, der für eine Masterarbeit angemessen ist, umsetzbar sind. Im Falle der Anforderung **F4.1** bestand beispielsweise das Risiko, dass ein Roslyn-Analyzer nicht die Einstellungen eines Plugins auslesen oder verändern kann, da das Roslyn SDK und Visual Studio SDK autark arbeiten. Standardmäßig wird dieser Vorgang auch nicht unterstützt. In Sharper Crypto-API Analysis wurde daher mittels Schnittstellen, darunter auch die Einstellungsschnittstelle `ISharperCryptoApiAnalysisSettings`, eine Brücke zwischen den auf dem Roslyn SDK basierenden Codeanalysen und dem IDE Plugin geschaffen. Diese Brücke wird als das Sharper Crypto-API Analysis SDK bezeichnet. Die Einstellungsschnittstelle beinhaltet unter anderem auch den Wert, der vom Benutzer festgelegten Analysehärte. Diese kann nun von einem Analyzer ausgelesen und verarbeitet werden. Als Resultat werden Codeanalysen basierend auf den Einstellungen unterschiedlich eingestuft und dem Benutzer gemeldet. Wann jedoch ein Analyzer ausgeführt wird, ist immer noch durch die internen Mechanismen in Visual Studio geregelt. Änderungen an den Einstellungen, die auch einen Analyzer betreffen, werden demnach erst vorgenommen, wenn Visual Studio den Analyzer neu ausführt. Beobachtungen zeigen, dass dies jedoch spätestens mit der Eingabe eines neuen Zeichens in einer Quelltextdatei geschieht. Neben dem Einstellen der Analysehärte können Analyzers auch abgeschaltet werden.

> **FF3** und **FF4** werden daher wie folgt beantwortet: Um Codemeldungen durch einen Roslyn-basierten Analyzer anzupassen, bedarf es einer Architektur, welche Analyzer und das SCAT miteinander verbindet. Über diese

[4] API Dokumentation: https://docs.microsoft.com/en-us/dotnet/api/index?view=visualstudiosdk-2017, Zugriff: 19.12.2018

[5] SDK Dokumentation: https://docs.microsoft.com/en-us/visualstudio/extensibility/visual-studio-sdk?view=vs-2017, Zugriff: 19.12.2918

[6] Nach der EU Richtlinie (Rat der Europäischen Union, 2009, S. 2) ist das Dekompilieren zum Zweck der Schaffung von Kompatibilität auch ohne Erlaubnis des Herstellers gestattet.

Architektur können Informationen ausgetauscht werden, die der Analyzer verarbeiten muss.

Um Analyseberichte mit unterschiedlicher Warnstufe zu erzeugen, muss der Analyzer ein Mapping durchführen, welches die Einstellungen des Nutzers auf die von Roslyn und Visual Studio benötigten Datentypen wie `DiagnosticSeverity` überführt. Je nach Nutzereinstellung kann die Meldung der Analyse auch unterdrückt werden. Somit lassen sich die Erkennungsraten durch den Nutzer anpassen. Welche False Positive Rate Sharper Crypto-API Analysis besitzt und ob sich diese durch die Nutzereinstellungen signifikant reduzieren lässt, war nicht Gegenstand dieser Arbeit und wurde daher nur oberflächlich untersucht (s. Abschnitt 5.4).

In wie weit ein Analyzer die Einstellungen des Nutzers interpretiert und umsetzt, kann durch das Sharper Crypto-API Analysis SDK nur empfohlen werden. Die Erweiterungsentwickler der Analyzer stehen hier in der Verantwortung diese Empfehlungen einzuhalten.

Durch die hier neu vorgestellten Funktionen kann der Benutzer nicht nur Codeanalysen konfigurieren, sondern auch weitere Extensions für das SCAT installieren. Während die Konfiguration eines Analyzers das Ziel hat, die False Positive Rate des SCAT zu regulieren, stellt die Erweiterbarkeit sicher, dass False Negatives durch weitere Codeanalysen eliminiert werden können. Neue hinzugefügte Extensions erlauben nicht nur eine Ausweitung von neuen Analysen für die .NET Krypto-API, sondern auch das Unterstützen weiterer API-Anbieter wie die Bouncy Castle Crypto API[7].

FF5 betrachtet sowohl die Seite der Benutzer, als auch die Seite von Erweiterungsentwicklern. Die Frage lässt sich jedoch in dieser Arbeit nicht absolut beantworten, unter anderem weil sich das SCAT noch im Entwicklungsstadium befindet. Die folgenden Absätze geben jedoch eine erste Einschätzung zur Beantwortung dieser Frage.

Für die Benutzererweiterbarkeit wurden anhand einer ausgiebigen Literaturrecherche in Kapitel 3 die Anforderungen **F1.2**, **F3.4**, **F4.1** und **F5** abgeleitet. Im speziellen Bezug auf die Codeanalysen der Nutzung von Krypto-APIs lag bei der Entwicklung dieses SCAT der Fokus auf individuelle Anpassbarkeit von Warnstufen. Die Erweiterbarkeit durch herunterladbare Extensions ist ebenfalls erforderlich, um dem Benutzer weitere Extension anzubieten, die in dieser Arbeit nicht berücksichtigt wurden. Insgesamt bietet Sharper Crypto-API Analysis jedoch nur ein Minimum an denkbaren Konfigurationen und Einstellungen, um die hier aufgestellten Anforderungen zu erfüllen.

Die Erweiterbarkeit durch Anwendungsentwickler wird durch das Sharper Crypto-API Analysis SDK sichergestellt. Die Planung eines SDK erfolgte bereits mit dem ersten Entwicklungstag des Projekts. Namensgebung der Assemblies und die Struktur der verfügbaren Namespaces orientieren sich stark an dem Visual Studio und Roslyn SDK. Entwicklern soll dadurch die Orientierung erleichtert werden. Beim Funktionsumfang des SDK wurde darauf geachtet, dass möglichst viele Funktionen des

[7] https://www.bouncycastle.org, Zugriff: 19.12.2018

SCAT, ähnlich wie auch bei Visual Studio Plugins, durch das SDK erweitert oder neu hinzugefügt werden können. Umfang und Komplexität der verfügbaren Komponenten sind hingegen recht simpel gehalten (beispielsweise die `SimpleTemplateEngine` oder eine `IWizardPage`). Zentraler Namespace dieses SDK ist `SharperCryptoApiAnalysis.Interop.CodeAnalysis`. Hier befinden sich alle notwendigen Schnittstellen und Implementationen zur Entwicklung neuer Analyzer. Trotz dieser vereinfachenden Maßnahmen, wird jedoch eine gewisse Erfahrung beim Umgang der Microsoft-APIs vorausgesetzt.

Bereits zu Beginn der Planungsphase von Sharper Crypto-API Analysis war die Erweiterbarkeit des Tools durch Assemblies, mittlerweile durch das Konzept der Extensions ersetzt, vorgesehen. Soweit dem Autor bekannt ist, gibt es kein Visual Studio Plugin, welches die Erweiterbarkeit seiner Funktionalitäten durch herunterladbare Assemblies selbst verwalten kann. Es stellte sich daher die Frage, ob Visual Studio die Selbstverwaltung seiner Plugins zulässt. Idealerweise wäre das Plugin in der Lage Extensions herunterzuladen, diese bei Visual Studio anzumelden, welches die Extension dann, falls erforderlich, in die Laufzeitumgebung lädt und zur Verwendung bereitstellt. Das größte Risiko ging von der Funktionsweise aus, wie Visual Studio Assemblies lädt. Würde die IDE die referenzierten Assemblies eines Plugins einlesen und nur diese laden, hätte dies zur Folge gehabt, dass das SCAT für jede Extension mit aktualisierten Assemblyreferenzen neu kompiliert werden müsste. Die Umsetzung eines solchen Verfahrens wäre im vorgegeben Zeitraum kaum möglich gewesen und über dies noch höchst impraktikabel. Erste Experimente in der frühen Entwicklungsphase zeigten jedoch, dass Visual Studio alle Assemblies lädt, die in einem bestimmten XML-Markup aufgelistet sind. Siehe hierzu Abschnitt 4.5.6. Die Erweiterbarkeit durch Hinzufügen neuer Assemblies konnte somit sichergestellt werden.

FF6 wird daher wie folgt beantwortet: In Sharper Crypto-API Analysis können Extensions heruntergeladen und installiert werden. Zur Verwendung dieser Extensions wird jedoch ein Neustart der IDE vorausgesetzt, damit diese durch das MEF geladen werden können. Ob Extensions, im Wesentlichen `.dll` Dateien, auch ohne Neustart in die IDE geladen werden können, kann diese Arbeit nicht zeigen. Dem Autor sind jedoch keine anderen Visual Studio Plugins bekannt, die ein derartiges Vorgehen umsetzten. Ebenfalls konnte in der Dokumentation des SDK keine Information zu diesem Thema entnommen werden. Prinzipiell bietet die CLR die Möglichkeit an, Assemblies zu einem späteren Zeitpunkt nachzuladen, jedoch ist nicht sichergestellt, ob MEF-Komponenten, wie sie Visual Studio verlangt, nachträglich korrekt in den MEF-Speicher der IDE eingebunden werden können. Der Zeit- und Arbeitsaufwand erschien dem Autor im Vergleich zum Vorteil, einen einzigen Neustart einzusparen, als zu hoch, um für diese Arbeit weiter in Betracht zu kommen.

5.4 Evaluierung der Codeanalysen

In diesem Abschnitt werden die durchgeführten Maßnahmen zum Testen der hier entwickelten Analyzer vorgestellt. Abschließend erfolgt eine kritische Bewertung der auffindbaren Fehler durch die Analyzer.

5.4.1 Modultests

Der Umgang mit dem Roslyn SDK zum Erstellen von Codeanalysen erfordert ein Grundverständnis der C#-Syntax und wie der Kompiler, also Roslyn, arbeitet. Eine Codeanalyse erfolgt, indem Quelltext zunächst vom Kompiler verarbeitet wird. Das daraus resultierende Modell, auch Syntaxbaum (eng.: syntax tree) genannt, gelangt dann zum Analyzer, der die Codeanalyse durchführt. Analyzer, die Visual Studio ausführt, erhalten die Syntaxbäume aus den Dateien des geöffneten Softwareprojekts der IDE. Diese Aktion führt die IDE automatisch aus. Eine Möglichkeit also Analyzers zu testen, wäre eine zweite Instanz der IDE zu öffnen, welche den zu testenden Analyzer ausführt. Diese zweite Instanz enthält ein Softwareprojekt, welches den Code implementiert, den der Analyzer überprüfen soll. Auch wenn dieses Testverfahren möglich ist, hat es den Nachteil eine Menge Zeit damit aufzubringen die zweite Instanz zu starten[8], das Projekt zu laden und die Codeanalyse durchzuführen. Um diesen Prozess zu beschleunigen, bietet Visual Studio eine Vorlage für Modultests von Analyzern. Der zu überprüfende Code wird als `string` Variable an einen Wrapper übergeben, der den Anaylzer ausführt. Die Notwendigkeit einer weiteren IDE Instanz und einem Testsoftwareprojekt entfallen damit. Das Listing 5.1 zeigt beispielhaft einen Modultest zur Analyse eines hartkodierten IV. Die Variable `test` enthält den zu überprüfenden Code. Die Variable `expected` enthält den erwarteten Analysebericht. Der Aufruf der Methode `VerifyCSharpDiagnostic` führt den Modultest aus. Ist der Analyzer richtig implementiert, wird eine Analysemeldung in der Codezeile 9 ausgegeben[9]. Während der Entwicklung von Sharper Crypto-API Analysis wurden für alle Analyzer derartige Modultests angefertigt. Um die Validität des Analyzers sicherzustellen, wurden viele verschiedene Codes, wie in der Variable `test` zu sehen, überprüft. Einige dieser Codes enthielten den Fehler andere wiederum nicht. Erst nachdem alle Modultests eines Analyzers positiv ausgegangen sind, wurde die Entwicklung dieses Analyzers als beendet betrachtet.

```
1 [TestMethod]
2 public void TestConstantIVByteArrayAssignment()
3 {
4     var test = @"using System.Security.Cryptography;
5         namespace SharperCryptoApiAnalysis.BaseAnalyzers.Tests
6         {
7             internal class DraftClass
8             {
9                 public static void TestMethod()
```

[8] Im Idealfall, sollte diese zweite IDE Instanz mit dem Debugger der ersten Instanz verbunden sein, um Funktionen wie Haltepunkte im Code des Analyzers zu unterstützen.

[9] Diese Zeilenangabe bezieht sich auch die Zeile des Codes in der `string` Variable, nicht auf die Zeile des Listings.

```
10                  {
11                      using (var aes = new AesManaged())
12                      aes.IV = new byte[]{12,12};
13                  }
14              }
15          }";
16
17      //Create the expected analysis report
18      var expected = new DiagnosticResult();
19
20      //Unit test will fail if actual report does not match the
           expected report
21      VerifyCSharpDiagnostic(test, expected);
22  }
```

Listing 5.1: Modultest für den Analyzer zum Überprüfen eines hartkodierten Initialisierungsvektors

5.4.2 Referenzprojekt „FalseCrypt“

Die Einsatz von Modultests kann in mit einem Laborversuch verglichen werden. Die Testumgebung ist abgesichert und durch äußere Einflüsse geschützt. Ob die Codeanalysen auch Fehler in einem Anwendungscode finden, kann daher durch Modultests nicht sichergestellt werden. Als ein potenzieller Störfaktor kann der ungewisse Datenfluss einer Variable gesehen werden. Die Modultests zeigen, dass die Analyzer in Sharper Crypto-API Analysis in der Lage sind, den Datenverlauf über mindestens zwei Variablen zurückzuverfolgen. Listing 5.2 zeigt dies an einem Beispiel: Die Methode `Foo` nimmt eine `int` Variable als Parameter. Der Parametername weist den Programmierer darauf hin eine zufällige Zahl zu benutzten. Der Analyzer `A` soll nun überprüfen, ob der Programmierer sich an diese Vorschrift hält, indem er keinen konstanten Wert als Parameter verwendet. Die bisher entwickelten Anaylzer können diesen Verlauf nachvollziehen und melden entsprechen einen Analysebericht.

```
1  int i = 0;
2  int j = i;
3  Foo(totallyRandomNumber: j); //Analyzer A knows j ist a
      constant variable
```

Listing 5.2: Datenfluss über zwei Variablen in C#

Das hier aufgeführte Beispiel zeigt einen Code, der keinen weiteren Einflüssen unterliegt und eignet sich daher für Modultests. In der Praxis kann der Datenwert der Variable `j` jedoch einen weitaus komplexeren Ursprung als durch die Variable `i` besitzen. Um das Verhalten von Sharper Crypto-API Analysis auch außerhalb der Modultests zu testen, wurde im Rahmen dieser Arbeit ein Referenzprojekt namens *FalseCrypt*[10] entwickelt. Im Gegensatz zu seinem Namensvetter *TrueCrypt*, verschlüsselt FalseCrypt lokale Dateien mit absichtlich falscher Verwendung der .NET Krypto-API. Anhand dieses Programms soll die Erkennungsrate Sharper Crypto-API Analysis

[10] https://github.com/AnakinSklavenwalker/FalseCrypt, Zugriff: 20.12.2018

festgestellt werden. Dank der Veröffentlichung des Quellcodes von FalseCrypt können auch andere Hersteller von SCATs ihre Tools überprüfen.

5.4.3 Eingrenzung und Bewertung

Bei der Entwicklung von Sharper Crypto-API Analysis konnten längst nicht alle möglichen Codeanalysen berücksichtigt werden. Um den Umfang für diese Arbeit übersichtlich und durchführbar zu halten, mussten einige Kompromisse eingegangen werden:

Der Schwerpunkt der Codeanalysen lag auf der Verwendung von passwortbasierten, symmetrischen Verschlüsselungsverfahren. Weitere Analyzer wurden nach verfügbarer Zeit und dem geschätzten Arbeitsaufwand umgesetzt. Codeanalysen, welche die asymmetrische Verschlüsselung betreffen, wurden nicht berücksichtigt.

Die Komplexität der implementierten Analysen ist auf ein gewisses Ausmaß begrenzt. Die Programmiersprache C# erlaubt dem Programmierer gewisse Operationen mit verschiedenen syntaktischen Ausdrücken umzusetzen. Hier das Beispiel zum Initialisieren eines neuen Bytearrays: `new[]{(byte)123}` ≡ `new byte[]{123}`. Auch wenn beide Varianten ähnlich aussehen, so sind sie für den Kompiler zwei verschiedene syntaktische Konstrukte. Ein Analyzer, der feststellen will, ob die Erzeugung eines Arrays zur Kompilierungszeit einen konstanten Wert besitzt, muss beide Varianten unterstützten. In Sharper Crypto-API Analysis wurden die Analyzer in einem Umfang entwickelt, dass sie einige dieser Fälle berücksichtigen. An anderen Stellen wird jedoch vorausgesetzt, dass sich der Programmierer an gängige Codekonventionen der Sprache C# hält. Ein Beispiel, welches von Sharper Crypto-API Analysis derzeit nicht unterstützt wird, zeigt Listing 5.3. Die Methoden `DeriveKey` und `DeriveKey2` leiten jeweils einen Schlüssel von dem Passwort `"123"` ab. Das Salt ist in beiden Fällen 8 Bytes lang. Ein Analyzer im aktuellen Entwicklungsstadium kann jedoch nicht feststellen, dass der Ausdruck `int.Parse("8")` in jedem Fall den konstanten Wert `8` erzeugt. Eine Codeanalyse die feststellen soll, ob dass Salt ausreichend groß ist, liefert demnach in der Methode `DeriveKey2` ein False Negative.

```
1  // Analyzer knows salt size is 8 bytes
2  public static byte[] DeriveKey()
3  {
4      using (var pbkdf = new Rfc2898DeriveBytes("123", 8))
5          return pbkdf.GetBytes(16);
6  }
7
8  // Analyzer does NOT know salt size is 8 bytes
9  public static byte[] DeriveKey2()
10 {
11     using (var pbkdf = new Rfc2898DeriveBytes("123",
           int.Parse("8")))
12         return pbkdf.GetBytes(16);
13 }
```

Listing 5.3: Methoden mit Schlüsselableitungsfunktionen und konstantem Salt

Ferner wurden in dieser Arbeit lediglich Codeanalysen der Nutzung von passwortbasierten, symmetrischen Verschlüsselungsverfahren betrachtet, welche, basierend auf der Literaturrecherche, die gängigsten Fehler von Programmierern abdecken. Analyzer, die erkennen, ob Klassen der Schnittstelle `IDisposable` wie `Aes` korrekt geschlossen werden, wurden nicht implementiert. Die untersuchte Literatur, darunter auch Tutorials und Beispielcodes gaben hierfür keinen Anlass.

Ein weiterer Kompromiss der eingegangen werden musste, war die Rückstellung der Implementation von Quick-Fixes. Grund ist der zusätzliche Arbeitsaufwand, der erforderlich ist. Quick-Fixes in Roslyn werden identisch wie ein Analyzer programmiert. Daher müssen auch hier sämtliche syntaktische Ausdrücke, wie oben beschrieben, einzeln betrachtet werden. Demonstrativ wurde in Sharper Crypto-API Analysis ein Quick-Fix umgesetzt, der die Zuweisung eines konstanten IV (s. **SCAA001**) behebt. Anhand der Implementation dieses Quick-Fix konnte abgeschätzt werden, dass die Entwicklungszeit eines Quick-Fixes in etwa mit der des passenden Analyzers gleichzusetzen ist.

Es bleibt festzuhalten, dass die implementierten Codeanalysen zur Nutzung von symmetrischen Verschlüsselungsverfahren in Sharper Crypto-API Analysis bereits das von Programmierern am häufigsten verwendete kryptographische Verfahren (s. Tabelle 3.1) abdeckt. Die Analysen versuchen dabei so viele Fehler im Code aufzudecken, ohne jedoch einerseits den Entwickler zu stark einzuschränken, was beim White-List Analyseverfahren passieren kann, andererseits konnten auch nicht alle Eventualitäten abgedeckt werden, sodass False Negative Fälle trotzdem möglich sind.

Zum Abschluss dieser Arbeit wurde Sharper Crypto-API Analysis anhand des Referenzprojekts FalseCrypt getestet und mit dem SCAT *Sonarlint* (s. Abschnitt 2.3) verglichen. In FalseCrypt wurden insgesamt 21 dokumentierte Fehler der Nutzung der .NET Krypto-API integriert. Diese lassen sich wie folgt gruppieren:

- Semantischer Fehler (1 Fehler vorhanden): Zu dieser Gruppe wird ein Fehler in FalseCrypt gezählt, bei dem der Verschlüsselungsschlüssel von einem Passwort nur einmal für mehrere Verschlüsselungsoperationen abgeleitet wird. Die darauf resultierenden verschlüsselten Daten haben demnach alle das gleiche Salt.

- Fest kodierte Zugangsdaten (1 Fehler vorhanden): Zum Starten des Programms wird ein Passwort verlangt. Dessen MD5 Hashwert ist in einer Variable fest im Quelltext kodiert. Mit Hilfe von Onlinediensten[11] können Hashwerte und ihre passenden Urbilder in Datenbanken einfach gefunden werden.

- Fehler im Umgang mit einem Zufallsgenerator (3 Fehler vorhanden): In FalseCrypt wird der Zufallsgenerator `System.Random` mit konstantem Seed zum Generieren eines Salt benutzt.

- Verwendung schwacher Hashingverfahren (1 Fehler vorhanden): FalseCrypt benutzt MD5 als Standardhashingverfahren.

- Schwache Anwendung von Schlüsselableitungsfunktionen (2 Fehler vorhanden): Die Schlüsselableitfunktion PBKDF2 in FalseCrypt wird mit einem zu kurzem Salt und zu wenigen Iterationen verwendet.

[11] https://crackstation.net, Zugriff: 28.12.2018

- Schwache Nutzung einer symmetrischen Verschlüsselung (6 Fehler vorhanden): Das symmetrische Verschlüsselungsverfahren in FalseCrypt ist DES mit dem Betriebsmodus ECB. Der IV, auch wenn er für ECB nicht benötigt wird, der dem Verschlüsselungsverfahren zugewiesen wird, ist konstant.

- Fehler beim Freigeben von Speicherressourcen (7 Fehler vorhanden): In dieser Gruppe befinden sich Fehler, bei welchen Speicherressourcen nicht ordnungsgemäß freigegeben werden. Dazu gehören das Leeren von Arrays, Löschen von Zeichenketten aus dem Speicher oder das Aufrufen der `Dispose` Methode.

Die Tests der SCATs wurden unter gleichen Bedingungen durchgeführt. Installierte Plugins, darunter auch das Tool *Resharper*, wurden deaktiviert, sodass nur das zu untersuchende SCAT Codeanalysen durchführen konnte. Jeder Test erfolgte auf der selben Quellcodeversion von FalseCrypt[12]. Die SCATs wurden auf ihre Standardeinstellungen zurückgesetzt. Für Sharper Crypto-API Analysis wurde zusätzlich ein Test mit höchstmöglicher Analysehärte (Härteeinstelltung: *Strict*) durchgeführt[13]. Es wurden nur die Codeanalysen für den Vergleich betrachtet, die im Fehlerfenster der IDE eine Meldung erzeugt haben. Die Warnstufe (Fehler, Warnung, Mitteilung) wurde nicht differenziert bewertet. Die Tabelle 5.1 zeigt die Auswertung der durchgeführten Tests.

Bevor jedoch die Ergebnisse des Tests vorgestellt werden, soll hier der Hinweis erfolgen, dass eine abschließende Evaluierung von Sharper Crypto-API Analysis nicht allein durch die Analyse des Quelltexts von FalseCrypt erfolgen kann. FalseCrypt besitzt einerseits den Nachteil vom selben Entwickler wie Sharper Crypto-API Analysis programmiert zu sein. Daher kann angenommen werden, dass der Stil des Codes in beiden Projekten identisch ist. Der Quelltext von FalseCrypt kann daher auf die Funktionsweise der Analyzer angepasst sein. Andererseits kann der Quellcode von FalseCrypt nicht alle denkbaren Fehler enthalten, da das Programm trotz alledem noch lauf- und funktionsfähig sein soll. Die Kombination mehrerer schwacher Verwendungen der .NET Krypto-API ist daher nicht immer möglich. Die Codeanalyse von weiteren Softwareprojekten durch Sharper Crypto-API Analysis ist daher zu empfehlen.

Zum Zeitpunkt der Abgabe dieser Arbeit konnte Sharper Crypto-API Analysis mit den Standardeinstellungen elf von 21 Fehlern in FalseCrypt identifizieren. Die Anpassung der Analysehärte auf die höchste Stufe konnte einen Fehler mehr finden und erreichte somit eine True Positive Rate von 52,3%. Sonarlint hingegen konnte nur zwei Fehler finden. False Positives konnten bei beiden SCATs nicht festgestellt werden. Dies ist allerdings kein Indikator, dass keine False Positives existieren. Listing 5.4 zeigt einen Codeausschnitt, der im aktuellen Entwicklungsstand eine False Positive Meldung erzeugen würde. Der Grund ist, dass der zuständige Analyzer nicht erkennen kann, dass der Rückgabewert `8` in Zeile 11 nie erreicht werden kann. Die Methode `GetSaltSize` liefert demnach eigentlich immer einen für den Analyzer konformen Wert (`16`) zurück.

[12]Link zum Commit, mit welchem die Tests durchgeführt wurden: https://github.com/AnakinSklavenwalker/FalseCrypt/tree/f761c77c498cae35d354c034e386720e4ed39c17, Zugriff: 01.01.2019

[13]Die Tests von Sharper Crypto-API Analysis wurden auf der Codebasis dieses Commits durchgeführt: https://github.com/AnakinSklavenwalker/SharperCryptoApiAnalysis/tree/34518a4f4ac1a018f4068da4d658fdfd57a83d31, Zugriff: 01.01.2019

Mit aktuellen 47% bis 52% liegt die True Positive Rate von Sharper Crypto-API Analysis bei einem Wert, der allgemein von SCATs erreicht wird. (Tomassi, 2018; Baca et al., 2008; Chelf & Ebert, 2009). Das Ergebnis wird jedoch derzeit von Fehlern in FalseCrypt getrübt, die das SCAT noch nicht unterstützt. Darunter fallen auch alle sieben Fehler, die in die Fehlergruppe Freigabe von Speicherressourcen fallen. Analyzer, die die korrekte Anwendung des `IDisposable` Pattern überprüfen, existieren bereits und können für Sharper Crypto-API Analysis übernommen werden. Sobald die Codeanalyse dieser Fehler implementiert ist, könnte hier eine True Positive Rate von 78,6%, ähnlich wie bei CongiCrypt (Krüger et al., 2018, S. 8), erreicht werden.

Tabelle 5.1: Erkennungsrate von Sharper Crypto-API Analysis zum Zeitpunkt der Abgabe dieser Arbeit anhand der Codeanalyse von FalseCrypt

Messung	Sharper Crypto-API Analysis (Default)	Sharper Crypto-API Analysis (Strict)	Snoarlint
True Positive Rate	47,6% (10/21)	52,3% (11/21)	9,5% (2/21)
False Negative Rate	52,4%	47,7%	90,5%

```
1  public static void DeriveKey()
2  {
3      using (var pbkdf = new Rfc2898DeriveBytes("123",
           GetSaltSize()))
4      { }
5  }
6
7  //The Analyzer will identify that a least one return value
       results in a too short salt size
8  public static int GetSaltSize()
9  {
10     if (false)
11         return 8;
12     return 16;
13 }
```

Listing 5.4: Code, der derzeit in Sharper Crypto-API Analysis eine False Positive Meldung erzeugt

Kapitel 6

Ausblick und Fazit

In dieser Arbeit wurde die Entwicklung eines statischen Codeanalyseprogramms zur Identifizierung von Schwachstellen bei der Verwendung der Krypto-API aus dem .NET Standard 2.0 behandelt. Bevor in diesem Kapitel die wichtigsten Erkenntnisse noch einmal zusammengefasst werden, soll ein kurzer Ausblick auf zukünftige Forschungsschwerpunkte mit dem Thema beschrieben werden.

Mit der Entwicklung von *Sharper Crypto-API Analysis* sind noch nicht alle offenen Fragen beantwortet und Forschungsmöglichkeiten ergründet. In zukünftigen Arbeiten könnten die hier erzielten Ergebnisse für die folgenden Zwecke weiterverwendet werden:

1. Erweiterung durch zusätzliche Analysers. Die Entwicklung eines voll ausgeprägten SCAT ist im Rahmen einer Masterarbeit nicht möglich. Daher wurde sich hier auf einige der wichtigsten Funktionalitäten und Analyzer beschränkt. Die erweiterbare Architektur von Sharper Crypto-API Analysis erlaubt es jedoch, zusätzliche Analysers einzufügen. In fortführenden Projekten könnte dadurch die True Positive Rate des SCAT erhöht werden.

2. Erweiterung von Einstellungsmöglichkeiten. Die in Kapitel 3 vorgestellten Ergebnisse machen deutlich, dass die Nutzererfahrung für das Ergebnis der Fehlererkennung mit einem SCAT verantwortlich ist. Durch geeignete Einstellungsmöglichkeiten können die Anwender das Tool nach ihren Bedürfnissen und Fähigkeiten anpassen. Da sich die Entwicklung auf die Zielgruppe der Programmierer konzentrierte, sind feinere Einstellungen, welche gerade für Sicherheitsexperten oder einem Qualitätssicherungsteam wichtig sind, noch nicht vorhanden. Bei der Weiterentwicklung des Tools kann auf diese speziellen Bedürfnisse dieser Zielgruppe eingegangen werden.

3. Portierung auf weitere Plattformen. Als Plattform für Sharper Crypto-API Analysis wurde die IDE Visual Studio gewählt. Wie sich aus den hier vorgestellten Untersuchungen ableiten lässt, hängt die Qualität der Software auch von der Vertrautheit des Entwicklers mit seinen Werkzeugen zusammen. Es erscheint daher logisch anzunehmen, dass ein SCAT für weitere IDEs verfügbar sein sollte. Als die am naheliegendste Alternative, wäre hier der Editor Visual

Studio Code zu nennen. Dieser erfuhr in den letzten Jahren einen kontinuierlichen Anstieg in seiner Beliebtheit[1][2].

4. Leistungs- und Erkennungsratenuntersuchungen. Nachdem in dieser Arbeit das Grundgerüst für ein spezialisiertes SCAT entwickelt wurde, könnten weitere Arbeiten die Einsatzfähigkeit unter Produktionsbedingungen untersuchen. Ein direkter Vergleich mit weiteren SCATs ist in diesem Kontext ebenfalls möglich und kann weitere Stärken und Verbesserungspotentiale von Sharper Crypto-API Analysis aufzeigen.

5. Testen der Gebrauchstauglichkeit. Einhergehend mit dem vorherigem Punkt, kann auch ein ausführliches Testen der Gebrauchstauglichkeit mit Entwicklern Aufschluss über die Leistungsfähigkeit geben. Ebenfalls wäre der Vergleich zu anderen SCATs interessant. Besonderer Fokus könnte auf den Fragen liegen, inwiefern die angezeigten Erklärungen zum Verständnis des Entwicklers beitragen.

6. Integration von CrySL. In dieser Arbeit unberücksichtigt ist die Regelsprache CrySL, mit welcher CogniCrypt Codeanalysen durchführt. CrySL ermöglicht durch seine Syntax prinzipiell einen programmiersprachenunabhängigen Einsatz. In zukünftigen Projekten könnte eine Implementierung einer auf C# basierenden CrySL-Engine erfolgen, die dann in Sharper Crypto-API Analysis integriert werden kann. Hier verspricht sich auch die Möglichkeit neue Analysen zur Laufzeit von Visual Studio zu integrieren, ohne dass die IDE neugestartet werden muss.

Im zweiten Kapitel wurden zunächst Grundlagen über Begriffe der Themen Krypto-API und statische Codeanalyseprogramme vorgestellt. Das dritte Kapitel ging dann unter Berücksichtigung der Fachliteratur detaillierter auf diese Begriffe ein und erarbeitete Ursachen, Wirkung und Lösungsansätze von Schwierigkeiten bei der Benutzung von Krypto-APIs. Als ein bereits vorhandenes SCAT, welches sich ausschließlich der Nutzung von Krypto-APIs widmet, konnte CogniCrypt gefunden werden, welches demnach auch als Vorbild für das hier entwickelte SCAT diente. Anhand von abschließenden Forschungsfragen erarbeitet das dritte Kapitel somit auch die Existenzgrundlage dieser Arbeit. Das vierte Kapitel widmet sich ganz dem Entwicklungsprozess eines neuen SCAT für die Programmiersprache C#. Zunächst wurden die Anforderungen sowie verfügbaren Codeanalysen formuliert, die dieses Tool erfüllen soll. Es folgt eine ausführliche Dokumentation der entwickelten Komponenten mit ihren Eigenschaften.

Als Resultat dieser Arbeit entstand das statische Codeanalysetool Sharper Crypto-API Analysis für die IDE Visual Studio 2017. Die Notwendigkeit dieses Tools ergibt sich aus den folgenden Problemfeldern:

- Die Untersuchung von Quelltexten offenbart, dass die meisten Programmierer nicht in der Lage sind, die vorhandenen Krypto-APIs fehlerfrei zu verwenden.

[1] Top IDE Index: https://pypl.github.io/IDE.html, Zugriff: 30.12.2018

[2] Stackoverflow Developer Survey: https://insights.stackoverflow.com/survey/2018/#technology-most-popular-development-environments, Zugriff: 30.12.2018

- Die Entwickler kennen offenbar nicht die einzelnen kryptographischen Verfahren und ihre Verwundbarkeiten. Als Ausweg kopieren und verwenden sie mitunter veraltete oder schlechte Tutorials, die ihrerseits Fehler beinhalten und schwache kryptographische Verfahren benutzten.
- Die Überprüfung der Sicherheit von Software wird in vielen Entwicklungsprozessen vernachlässigt, obwohl automatisierte Qualitätssicherungsmaßnahmen in der Lage sind, beim Identifizieren und Beheben von falschen Verwendungen der Krypto-APIs zu helfen.
- Insgesamt gibt es nur wenige programmbasierte Lösungen zur Untersuchung dieser Fehler. Eines der Programme ist CogniCrypt.
- Die Programmiersprache C# sowie die .NET Plattformen sind in aktuellen Forschungen unterrepräsentiert, sodass eine Ungewissheit über potenzielle Parallelen zur Programmierung mit Java bestehen.

Das Primärziel von Sharper Crypto-API Analysis ist all diese Problemfelder anzusprechen und den Entwicklern bei der Verwendung der .NET Krypto-API zu unterstützen. Dadurch sollen Bedrohungen und Schwachstellen in Softwareanwendungen vermeiden werden. Um dieses Ziel zu erfüllen, wurden vier elementare Funktionsgruppen implementiert.

1. **Codeanalyse:** Die Codeanalyse ist der zentrale Baustein eines SCAT. Die meisten SCATs konzentrieren sich dabei lediglich auf die bekannten Beispiele wie Pufferüberläufe. Dieses SCAT hingegen betrachtet ausschließlich die Verwendung der .NET Standard 2.0 Krypto-API. Um den Umfang für diese Arbeit zu reduzieren, wurden die verfügbaren Analysen jedoch vorrangig auf die Benutzung von symmetrischen, passwortbasierten Verschlüsselungsverfahren begrenzt.
2. **Aufgabenorientierte Codegenerierung:** Eine häufige Ursache, warum Entwickler Krypto-APIs falsch verwenden ist, weil sie nicht wissen, wie sie eine aufgabenbezogene Anforderung mittels der API umsetzten können. Die Codegenerierung dieses Tools erlaubt es dem Programmierer, mittels eines Assistenten, Aufgaben auszuwählen. Anhand von vorher festgelegten Vorlagen, wird dann der passende Code zu dieser Aufgabe generiert und dem Softwareprojekt hinzugefügt.
3. **Konfiguration:** In manchen Fällen ist es für den Programmierer übersichtlicher Codeanalysen anders darzustellen oder ganz zu unterdrücken. Mittels einiger Einstellungsoptionen lässt sich das Verhalten der Codeanalysen anpassen. Ebenfalls können Einstellungswerte auf Git-Repositorys gespeichert werden.
4. **Erweiterbarkeit:** Durch die Unterstützung von sogenannten Extensions, kann der Benutzer des SCAT von weiteren Anbietern neue Codeanalysen und Codegenerierungen beziehen.

Die Umsetzung dieser Funktionsgruppen erfolgte teils durch neuartige Implementierungsverfahren, die auch aufzeigen wie fortgeschritten und erweiterbar die IDE Visual

Studio ist. Zu nennen ist hier die Realisierung der Erweiterbarkeit des SCAT.

Ein ganz besonderer Fokus von Sharper Crypto-API Analysis liegt darauf, den Programmierer auf Schwachstellen in seinem Quelltext zu sensibilisieren und ihm das Wissen zu vermitteln, warum das SCAT eine Stelle im Code als Schwäche meldet. Aus diesem Grund besitzt Sharper Crypto-API Analysis die Funktion, dem Programmierer detaillierte Analyseberichte anzeigen zu lassen. Diese Berichte geben einerseits nützliche Informationen und Informationsquellen an, die zu Rate gezogen werden können und andererseits zeigen sie dem Programmierer, welche Folgen seine schwacher Code haben kann und wie er schnell und einfach zu beheben ist. Ihr Informationsgehalt ist daher wesentlich höher als der einer einfachen Fehlernummer, die mit einem Satz beschrieben wird. Auch die Codegenerierungen können helfen, dem Programmierer Wissen zu vermitteln. Indem die diese Funktionalität anhand von einfachen, aber ausführlich beschriebenen Fragen eine Krypto-Aufgabe implementiert, kann der Programmierer wichtige Merkmale der Aufgabe ablesen.

Die Entwicklungsphase von Sharper Crypto-API Analysis konnte zudem einige Stärken und Schwächen beim Verwenden der C#- und .NET-Technologien aufdecken. Die .NET Krypto-API ist durch eine einfachere Namensgebung ihrer Datentypen und einer simpleren Struktur im Gegensatz zur JCA leichter zu verstehen und damit womöglich einfacherer zu verwenden. Beim Entwickeln einer Roslyn-unterstützten Erweiterung für Visual Studio ist hingegen eine gewisse Erfahrung seitens des Entwicklers erforderlich. Die SDKs besitzen zwar eine Dokumentation, die alle verfügbaren Datentypen und Methoden beschreibt, jedoch fehlen an vielen Stellen wichtige Hinweise auf Verhaltensweisen oder andere hilfreiche Informationen. Die von Microsoft verfügbaren Codebeispiele des SDK sind des Weiteren zu kurz oder unvollständig, um von ihnen komplexere Funktionalitäten abzuleiten. Die Entwicklung erfordert daher, gerade für unerfahrene Programmierer, einen hohen Aufwand, Machbarkeitstests durchzuführen.

Mit der Veröffentlichung von Sharper Crypto-API Analysis auf GitHub[3] und der besonderen Berücksichtigung eine erweiterbare Programmstruktur durch umfangreiche und gut dokumentierte Schnittstellen bereitzustellen, stehen allen interessierten Entwicklern die Möglichkeiten bereit, aktiv an der Weiterentwicklung mitzuwirken, um letzten Endes das Ziel zu erreichen, dass unerfahrene Programmierer weniger Fehler beim Verwenden von Krypto-APIs produzieren.

[3]https://github.com/AnakinSklavenwalker/SharperCryptoApiAnalysis, Zugriff: 30.12.2018

Literaturverzeichnis

Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L. & Stransky, C. (2017). Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)* (S. 154–171). doi:10.1109/SP.2017.52. (Siehe S. 22, 37)

Ammann, P. & Offutt, J. (2008). *Introduction to Software Testing* (1. Aufl.). New York, NY, USA: Cambridge University Press. (Siehe S. 13).

Anderson, R. J. (1994). Why Cryptosystems Fail. (Bd. 37, *11*, S. 32–40). ACM. (Siehe S. 21).

Aniche, M. F. & Gerosa, M. A. (2010). Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops* (S. 469–478). doi:10.1109/ICSTW.2010.16. (Siehe S. 20)

Arkin, B., Stender, S. & McGraw, G. (2005). Software Penetration Testing. *IEEE Security & Privacy*, *3*(1), 84–87. (Siehe S. 15, 16).

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J. & Pugh, W. (2008). Using Static Analysis to Find Bugs. *IEEE Software*, *25*(5). (Siehe S. 27, 37).

Baca, D., Carlsson, B. & Lundberg, L. (2008). Evaluating the Cost Reduction of Static Code Analysis for Software Security. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (S. 79–88). PLAS '08. doi:10.1145/1375696.1375707. (Siehe S. 25, 26, 28, 46, 93)

Baca, D., Petersen, K., Carlsson, B. & Lundberg, L. (2009). Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? In *2009 International Conference on Availability, Reliability and Security* (S. 804–810). IEEE. (Siehe S. 10, 25–28, 37).

Bacchelli, A. & Bird, C. (2013). Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering* (S. 712–721). doi:10.1109/ICSE.2013.6606617. (Siehe S. 25)

Baker, R. A., Jr. (1997). Code Reviews Enhance Software Quality. In *Proceedings of the 19th International Conference on Software Engineering* (S. 570–571). ICSE '97. doi:10.1145/253228.253461. (Siehe S. 10)

Bardas, A. G. (2010). Static Code Analysis. *Romanian Economic Business Review*, *4*(2), 99–107. Zugriff unter https://ideas.repec.org/a/rau/journl/v4y2010i2p99-107.html. (Siehe S. 48)

Barker, E. B. (2016). Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms. *NIST special publication*, *800-175B*, 67. (Siehe S. 56).

Barker, E. B. & Roginsky, A. L. (2015). *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. doi:10.6028/nist.sp.800-131ar1. (Siehe S. 59)

Barker, E., Feldman, L. & Witte, G. (2017). Guidance on the TDEA Block Ciphers. *ITL Bulletin*. (Siehe S. 56).

Basili, V. R. & Selby, R. W. (1987). Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, *SE-13*(12), 1278–1296. doi:10.1109/TSE.1987.232881. (Siehe S. 17)

Benington, H. D. (1983). Production of Large Computer Programs. *Annals of the History of Computing*, *5*(4), 350–361. doi:10.1109/MAHC.1983.10102. (Siehe S. 14)

Bertolino, A. (2007). Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07)* (S. 85–103). doi:10.1109/FOSE.2007.25. (Siehe S. 17)

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., ... Engler, D. (2010). A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, *53*(2), 66–75. (Siehe S. 27, 37, 46).

Bloch, J. (2014). A Brief, Opinionated History of the API. Zugriff 18. Oktober 2018 unter http://goo.gl/WLKLQc. (Siehe S. 9)

Bradner, S. (1997). *Key words for use in RFCs to Indicate Requirement Levels* (RFC Nr. 2119). RFC Editor. RFC Editor. Zugriff unter https://www.ietf.org/rfc/rfc2119.txt. (Siehe S. 106)

Braga, A., Dahab, R., Antunes, N., Laranjeiro, N. & Vieira, M. (2017). Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)* (S. 170–181). doi:10.1109/ISSRE.2017.27. (Siehe S. 27)

Chatzikonstantinou, A., Ntantogian, C., Karopoulos, G. & Xenakis, C. (2016). Evaluation of Cryptography Usage in Android Applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS)* (S. 83–90). BICT'15. doi:10.4108/eai.3-12-2015.2262471. (Siehe S. 22, 37)

Chelf, B. & Ebert, C. (2009). Ensuring the Integrity of Embedded Software with Static Code Analysis. *IEEE Software*, *26*(3), 96–99. doi:10.1109/MS.2009.65. (Siehe S. 10, 25, 26, 49, 93)

Chen, L. (2008). SP 800-108. Recommendation for Key Derivation Using Pseudorandom Functions (Revised). *NIST special publication*. (Siehe S. 54).

Chess, B. & McGraw, G. (2004). Static Analysis for Security. *IEEE Security & Privacy*, *2*(6), 76–79. (Siehe S. 10, 25–27, 49).

Chess, B. & West, J. (2007). *Secure Programming with Static Analysis* (First). Addison-Wesley Professional. (Siehe S. 26, 37, 43, 46, 49).

Clarke, S. (2004). Measuring API Usability. Zugriff 24. Oktober 2018 unter http://www.drdobbs.com/windows/measuring-api-usability/184405654. (Siehe S. 39)

Cova, M., Kruegel, C. & Vigna, G. (2010). Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th*

International Conference on World Wide Web (S. 281–290). WWW '10. doi:10.1145/1772690.1772720. (Siehe S. 28)

Daka, E. & Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (S. 201–211). doi:10.1109/ISSRE.2014.11. (Siehe S. 16, 17, 20, 39)

Diffie, W. & Hellman, M. E. (1977). Special feature exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, *10*(6), 74–84. (Siehe S. 51).

Dworkin, M. (2001). *Recommendation for block cipher modes of operation : Methods and Techniques*. doi:10.6028/nist.sp.800-38a. (Siehe S. 55, 56)

Eckert, C. (2013). *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. Walter de Gruyter. (Siehe S. 2).

ECMA International. (2012). *Standard ECMA-335 - Common Language Infrastructure (CLI)* (6. Aufl.). Zugriff 1. Januar 2019 unter https://www.ecma-international.org/publications/standards/Ecma-335.htm. (Siehe S. 13)

Egele, M., Brumley, D., Fratantonio, Y. & Kruegel, C. (2013). An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (S. 73–84). CCS '13. doi:10.1145/2508859.2516693. (Siehe S. 3, 22, 37, 51, 56)

Ehrsam, W. F., Meyer, C. H., Smith, J. L. & Tuchman, W. L. (1978). Message verification and transmission error detection by block chaining. US Patent 4,074,066. Google Patents. (Siehe S. 2).

Ellis, B., Stylos, J. & Myers, B. (2007). The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering* (S. 302–312). IEEE Computer Society. (Siehe S. 82).

Emanuelsson, P. & Nilsson, U. (2008). A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, *217*, 5–21. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). doi:https://doi.org/10.1016/j.entcs.2008.06.039. (Siehe S. 27)

Engebretson, P. (2013). *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Elsevier. (Siehe S. 15).

Ersoy, E. & Sözer, H. (2016). Extending Static Code Analysis with Application-Specific Rules by Analyzing Runtime Execution Traces. In *Computer and Information Sciences* (S. 30–38). Springer. Springer International Publishing. (Siehe S. 25).

Frauenhofer, S. (2013). Trend-und Strategiebericht: Entwicklung sicherer Software durch Security by design. SIT Technical Reports. Stuttgart: Frauenhofer Verlag. (Siehe S. 19).

George, B. & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology*, *46*(5), 337–342. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003. doi:https://doi.org/10.1016/j.infsof.2003.09.011. (Siehe S. 16, 20)

Grande, M. (2014). Anforderungen. In *100 Minuten für Anforderungsmanagement: Kompaktes Wissen nicht nur für Projektleiter und Entwickler* (S. 37–45). doi:10.1007/978-3-658-06435-8_7. (Siehe S. 44)

Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., ... Theofanos, M. F. (2017). *Digital Identity Guidelines: Authentication*

and Lifecycle Management. National Institute of Standards und Technology. doi:10.6028/nist.sp.800-63b. (Siehe S. 58)

Heitkötter, H., Hanschke, S. & Majchrzak, T. A. (2012). Evaluating Cross-Platform Development Approaches for Mobile Applications. In *International Conference on Web Information Systems and Technologies* (S. 120–138). Springer. (Siehe S. 1).

Hicks, M. (2017). What is soundness (in static analysis)? The Programming Languages Enthusiast. Zugriff 21. Mai 2018 unter http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/. (Siehe S. 27)

Howard, M. & Lipner, S. (2003). Inside the Windows security push. *IEEE Security & Privacy*, *99*(1), 57–61. (Siehe S. 46).

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T. & Kuo, S.-Y. (2004). Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web* (S. 40–52). WWW '04. doi:10.1145/988672.988679. (Siehe S. 27)

ISO/IEC. (2018). Information technology – Security techniques – Information security management systems – Overview and vocabulary. (Siehe S. 1).

ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. (2008). *IEEE STD 12207-2008*, 1–138. doi:10.1109/IEEESTD.2008.4475826. (Siehe S. 40)

Johnson, B., Song, Y., Murphy-Hill, E. & Bowdidge, R. (2013). Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering* (S. 672–681). IEEE Press. (Siehe S. 25, 27, 28, 37, 46, 47, 49).

Johnson, S. C. (1978). Lint, a C Program Checker, Bell Telephone Laboratories Murray Hill. (Siehe S. 10).

Jones, C. (2008). *Applied Software Measurement: Global Analysis of Productivity and Quality* (3rd). McGraw-Hill Education Group. (Siehe S. 16).

Jones, G. & Prieto-Diaz, R. (1988). Building and managing software libraries. In *Computer Software and Applications Conference, 1988. COMPSAC 88. Proceedings., Twelfth International* (S. 228–236). IEEE. (Siehe S. 21).

Kaliski, B. (2000). *PKCS #5: Password-Based Cryptography Specification version 2.0* (RFC Nr. 2898). RFC Editor. RFC Editor. Zugriff 17. Dezember 2018 unter https://www.ietf.org/rfc/rfc2898.txt. (Siehe S. 54, 58)

Kelsey, J., Schneier, B. & Wagner, D. (1997). Related-key cryptanalysis of 3-WAY, Biham-DES,CAST, DES-X, NewDES, RC2, and TEA. In *International Conference on Information and Communications Security* (S. 233–246). Springer. (Siehe S. 53, 56).

Kleidermacher, D. N. (2008). Integrating Static Analysis into a Secure Software Development Process. In *Technologies for Homeland Security, 2008 IEEE Conference on* (S. 367–371). IEEE. (Siehe S. 25, 47).

Kleuker, S. (2018). *Grundkurs Software-Engineering mit UML: der pragmatische Weg zu erfolgreichen Softwareprojekten*. Springer-Verlag. (Siehe S. 14).

Kononenko, O., Baysal, O. & Godfrey, M. W. (2016). Code Review Quality: How Developers See It. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (S. 1028–1038). doi:10.1145/2884781.2884840. (Siehe S. 16)

Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., ... Kamath, R. (2017). CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (S. 931–936). ASE 2017. Urbana-Champaign, IL, USA: IEEE Press. (Siehe S. 20–22, 31, 35, 37, 39, 47).

Krüger, S., Späth, J., Ali, K., Bodden, E. & Mezini, M. (2018). CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)* (10:1–10:27). (Siehe S. 3, 22, 31, 37, 51, 93).

Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Rupp, A. & Schimmler, M. (2006). How to Break DES for BC 8,980. *SHARCS '06–Special-purpose Hardware for Attacking Cryptographic Systems*, 17–35. (Siehe S. 51).

Lazar, D., Chen, H., Wang, X. & Zeldovich, N. (2014). Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (S. 7). ACM. (Siehe S. 22, 37).

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80–83. (Siehe S. 14).

McIntosh, S., Kamei, Y., Adams, B. & Hassan, A. E. (2014). The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (S. 192–201). MSR 2014. doi:10.1145/2597073.2597076. (Siehe S. 10)

McLellan, S. G., Roesler, A. W., Tempest, J. T. & Spinuzzi, C. I. (1998). Building more usable APIs. *IEEE Software*, *15*(3), 78–86. doi:10.1109/52.676963. (Siehe S. 24)

Mendel, F., Pramstaller, N., Rechberger, C. & Rijmen, V. (2006). On the Collision Resistance of RIPEMD-160. In *International Conference on Information Security* (S. 101–116). Springer. (Siehe S. 54).

Meng, N., Nagy, S., Yao, D. (, Zhuang, W. & Argoty, G. A. (2018). Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. (Siehe S. 3, 22, 28).

Morris, R. & Thompson, K. (1979). Password security: A case history. *Communications of the ACM*, *22*(11), 594–597. (Siehe S. 54).

Myers, G. J., Sandler, C. & Badgett, T. (2011). *The Art of Software Testing* (3rd). Wiley Publishing. (Siehe S. 13, 14).

Nadi, S., Krüger, S., Mezini, M. & Bodden, E. (2016). "Jumping Through Hoops": Why do Java Developers Struggle with Cryptography APIs? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (S. 935–946). ACM. (Siehe S. 22, 24, 30, 37, 39).

Nagappan, N. & Ball, T. (2005). Static Analysis Tools As Early Indicators of Pre-release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering* (S. 580–586). ICSE '05. doi:10.1145/1062455.1062558. (Siehe S. 25)

Novak, J., Krajnc, A. & Žontar, R. (2010). Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO* (S. 418–422). (Siehe S. 10, 25, 26).

Novikov, A. S., Ivutin, A. N., Troshina, A. G. & Vasiliev, S. N. (2017). The approach to finding errors in program code based on static analysis methodology. In *2017 6th Mediterranean Conference on Embedded Computing (MECO)* (S. 1–4). doi:10.1109/MECO.2017.7977127. (Siehe S. 17)

Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M. & Gordon, M. (2002). What Programmers Really Want: Results of a Needs Assessment for SDK Documentation. In *Proceedings of the 20th Annual International Conference on Computer Documentation* (S. 133–141). SIGDOC '02. doi:10.1145/584955.584976. (Siehe S. 24)

Oechslin, P. (2003). Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Annual International Cryptology Conference* (S. 617–630). Springer. (Siehe S. 54, 58).

Osherove, R. (2015). *The art of unit testing*. MITP-Verlags GmbH & Co. KG. (Siehe S. 15).

Paar, C. & Pelzl, J. (2016). *Kryptografie verständlich*. doi:10.1007/978-3-662-49297-0. (Siehe S. 9)

Paetsch, F., Eberlein, A. & Maurer, F. (2003). Requirements engineering and agile software development. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* (S. 308–313). doi:10.1109/ENABL.2003.1231428. (Siehe S. 40)

Partsch, H. (2010). Begriffliche Grundlagen. In *Requirements-Engineering systematisch: Modellbildung für softwaregestützte Systeme* (S. 19–68). doi:10.1007/978-3-642-05358-0_2. (Siehe S. 44)

Peltier, T. R. (2013). *Information security fundamentals*. CRC Press. (Siehe S. 1).

Rat der Europäischen Union. (2009). Richtlinie 2009/24/EG über den Rechtsschutz von Computerprogrammen. *Amtsblatt der Europäischen Union*. (Siehe S. 85).

Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, *74*(2), 358–366. (Siehe S. 27).

Rivest, R. (1998). *A Description of the RC2(r) Encryption Algorithm* (RFC Nr. 2268). RFC Editor. RFC Editor. Zugriff 17. Dezember 2018 unter https://www.rfc-editor.org/rfc/rfc1654.txt. (Siehe S. 53)

Robillard, M. P. & DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, *16*(6), 703–732. doi:10.1007/s10664-010-9150-8. (Siehe S. 22, 24, 37, 39)

Rogaway, P. (1995). Problems with proposed IP cryptography. *Unpublished manuscript*. (Siehe S. 2).

Santha, M. & Vazirani, U. V. (1986). Generating quasi-random sequences from semi-random sources. *Journal of Computer and System Sciences*, *33*(1), 75–87. doi:10.1016/0022-0000(86)90044-9. (Siehe S. 55)

Shen, H., Fang, J. & Zhao, J. (2011). EFindBugs: Effective Error Ranking for FindBugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (S. 299–308). doi:10.1109/ICST.2011.51. (Siehe S. 27)

Siebler, F. (2014). *Design Patterns mit Java: Eine Einführung in Entwurfsmuster*. Carl Hanser Verlag GmbH Co KG. (Siehe S. 67).

Spitz, S., Pramateftakis, M. & Swoboda, J. (2011). Ziele und Wege der Kryptographie. In *Kryptographie und IT-Sicherheit: Grundlagen und Anwendungen* (S. 1–42). doi:10.1007/978-3-8348-8120-5_1. (Siehe S. 2)

Stylos, J., Faulring, A., Yang, Z. & Myers, B. A. (2009). Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (S. 119–126). doi:10.1109/VLHCC.2009.5295283. (Siehe S. 37, 39)

Sun, B., Shu, G., Podgurski, A. & Robinson, B. (2012). Extending Static Analysis by Mining Project-specific Rules. In *Proceedings of the 34th International Conference on Software Engineering* (S. 1054–1063). doi:10.1109/ICSE.2012.6227114. (Siehe S. 26)

Tassey, G. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology, RTI Project*, *7007*(011). (Siehe S. 25).

Tomassi, D. A. (2018). Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-world Bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (S. 980–982). ESEC/FSE 2018. doi:10.1145/3236024.3275439. (Siehe S. 93)

Tsipenyuk, K., Chess, B. & McGraw, G. (2005). Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security & Privacy*, *3*(6), 81–84. doi:10.1109/MSP.2005.159. (Siehe S. 19, 51)

Turan, M. S., Barker, E. B., Burr, W. E. & Chen, L. (2010). *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications.* National Institute of Standards und Technology. doi:10.6028/nist.sp.800-132. (Siehe S. 55, 58)

Uwano, H., Nakamura, M., Monden, A. & Matsumoto, K.-i. (2006). Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications* (S. 133–140). ETRA '06. doi:10.1145/1117309.1117357. (Siehe S. 16, 20, 39)

Viega, J. [J.], Bloch, J. T., Kohno, Y. & McGraw, G. (2000). ITS4: a static vulnerability scanner for C and C++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)* (S. 257–267). doi:10.1109/ACSAC.2000.898880. (Siehe S. 10, 25)

Viega, J. [John] & McGraw, G. (2011). *Building Secure Software: How to Avoid Security Problems the Right Way (Paperback) (Addison-Wesley Professional Computing Series)* (1st). Addison-Wesley Professional. (Siehe S. 2).

Wang, X., Feng, D., Lai, X. & Yu, H. (2004). Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive*, *2004*, 199. (Siehe S. 23).

Wang, X., Yin, Y. L. & Yu, H. (2005). Finding collisions in the full SHA-1. In *Annual international cryptology conference* (S. 17–36). Springer. (Siehe S. 51).

Wang, X. & Yu, H. (2005). How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques* (S. 19–35). Springer. (Siehe S. 51).

Ware, M. S. & Fox, C. J. (2008). Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools. In *Proceedings of the 2008 Workshop on Static Analysis* (S. 12–21). SAW '08. doi:10.1145/1394504.1394506. (Siehe S. 26)

WASC. (2013). *Static Analysis Tools Evaluation Criteria*. 1.0. Web Application Security Consortium. Zugriff 5. Dezember 2018 unter http://projects.webappsec.org/w/file/fetch/62389783/satec_manual.pdf. (Siehe S. 10, 11, 37, 46)

Wätjen, D. (2018). Hashfunktionen. In *Kryptographie: Grundlagen, Algorithmen, Protokolle* (S. 93–112). doi:10.1007/978-3-658-22474-5_6. (Siehe S. 51)

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. & Vouk, M. A. (2006). On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*, *32*(4), 240–253. doi:10.1109/TSE.2006.38. (Siehe S. 27)

Anhang A

Anforderungen für Sharper Crypto-API Analysis

Die Schlüsselwörter „MUSS“, „DARF NICHT“, „ERFORDERLICH“, „SOLL“, „VERBOTEN“, „NÖTIG“, „NICHT NÖTIG“, „SOLL NICHT“, „EMPFOHLEN“, „DARF“, „KANN“ und „OPTIONAL“ werden nach der deutschen Interpretation des RFC 2119 (Bradner, 1997) der Adfinis SyGroup[1] verwendet.

Anforderungen

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
	Funktionale Anforderungen			
F1	**Codeanalyse**			
F1.0	C# Quellcodeanalyse zur Identifikation fehlerhafter Nutzung von der .NET Krypto-API	Das Tool MUSS in der Lage sein, Fehler bei der Nutzung der .NET Krypto-API unter dem Namespace `System.Security.Cryptography` im Quellcode der Anwendung zu identifizieren. Alle Analysen müssen C# unterstützen. Das Ziel MUSS dabei sein, eine möglichst niedrige False Negative und False Positive Rate zu erreichen.	Implementiert	Codeanalyse Implementiert durch Roslyn. Evaluierung der Klassifizierung durch Modultests erfolgt.

[1] https://github.com/adfinis-sygroup/2119/blob/master/2119de.rst, Zugriff: 10.122018

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F1.1	Codeanalyse, trotz unfertigem Code	Das Tool MUSS in der Lage sein, Codeanalysen durchzuführen, obwohl der Code u.U. nicht kompiliert werden kann.	Implementiert	Implementiert durch Roslyn.
F1.2	Unterdrückung der Analyse	Das Tool MUSS dem Nutzer die Möglichkeit bieten, bestimmte Codeanalysen oder Codeanalysen an bestimmten Stellen zu unterdrücken.	Implementiert	Teilweise durch Visual Studio implementiert.
F1.3	Bereitstellen von Analyseberichten	Das Tool MUSS positiv Analysierte Codestellen in Form eines Bericht dem Nutzer zugänglich machen. Dieser Bericht SOLL den Schweregrad des Fehlers, Ort der Codestelle und aus dem Fehler resultierende Bedrohungen beinhalten. Der Bericht kann weitere Informationen wie Internetlinks enthalten.	Implementiert	
1.4	Auflistung verfügbarer Analysen und Berichte	Das Tool MUSS alle verfügbaren Codeanalysen und Berichte anzeigen, ohne dass diese derzeit in Code analysiert wurden.	Implementiert	

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F1.5	Warnstufen für Berichte	Das Tool SOLL Berichte, je nach Schweregrad, unterschiedlichen Warnstufen zuordnen.	Implementiert	Teilweise durch Roslyn implementiert.
F1.6	Export von Berichten	Das Tool SOLL Berichte, in einem Dokumentenformat (PDF) zusammenfassen und abspeichern.	Nicht Implementiert	
F1.7	Interoperabilität der Berichte	Das Tool KANN Berichte in einer Form aufbereiten, die es anderen, externen Programmen ermöglicht, diese zu verarbeiten.	Nicht Implementiert	
F2	**Quick-Fixes**			
F2.0	Bereitstellen von Quick-Fixes	Das Tool MUSS in der Lage sein zu positiv bewerteten Codestellen, Fehlerbehebungen in Form von Quick-Fixes zur Verfügung zu stellen.	Implementiert	Implementiert durch Roslyn.
F2.1	Vorschau des Quick-Fix.	Verfügbare Quick-Fixes SOLLEN durch eine Vorschau anzeigen, welcher Code in welchem Ausmaß geändert wird.	Implementiert	Implementiert durch Visual Studio.

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F2.2	Auswahl eines Quick-Fix.	Sind für eine Codeanalyse mehrere Quick-Fixes verfügbar, SOLL der Nutzer die Möglichkeit haben zwischen den verschiedenen Optionen zu wählen.	Implementiert	Implementiert durch Visual Studio.
F3	**Codegenerierung**			
F3.0	Aufgabenbasierte Codegenerierung	Das Tool MUSS in der Lage sein Codefragmente zu generieren, die kryptographische Aufgaben implementieren.	Implementiert	
F3.1	Codeintegration	Werden Codefragmente generiert MÜSSEN diese in die aktuelle Projektstruktur des Quellcodes integriert werden. Dazu gehören u.a. die Anpassung des Namespace, Klassennamen und Kompileroptionen.	Implementiert	
F3.3	Auflistung verfügbarer Aufgaben	Das Tool MUSS alle verfügbaren Codegenerierungsaufgaben auflisten können.	Implementiert	
F3.4	Konfigurierbarkeit der Generierung	Die Codegenerierung SOLL die Möglichkeit unterstützen, durch Einstellungsmöglichkeiten sich an die Bedürfnisse des Nutzers anzupassen.	Implementiert	

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F3.5	Benutzerassistent	Das Tool SOLL, wenn es die Aufgabe erfordert, dem Nutzer einen Assistenten anbieten, der ihm hilft verfügbare Konfigurationsmöglichkeiten durchzuführen.	Implementiert	
F4	**Konfigurierung**			
F4.0	Konfigurierbarkeit	Das Tool MUSS dem Nutzer die Möglichkeit anbieten sein Verhalten den Bedürfnissen des Nutzers anzupassen.	Implementiert	
F4.1	Konfiguration der Erkennungsraten	Durch eine Nutzerkonfiguration SOLLEN Codeanalysen unterschiedlich bewertet werden. Je nach Bewertung erfolgt eine Einordnung der Analyse in eine Warnstufe. Die Anwendung der neuen Konfiguration MUSS mit dem nächsten Analysevorgang erfolgen.	Implementiert	Teilweise durch Roslyn implementiert.

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F4.2	Konfiguration per Git-Repository	Das Tool SOLL die Möglichkeit besitzen Konfigurationsinformationen aus frei wählbarem Git-Repository zu beziehen. Ist ein Repository angegeben, MÜSSEN Möglichkeiten zur Synchronisation der Tooleinstellungen vorhanden sein. Das Repository MUSS von einem beliebigen Git-Anbieter (z.B. GitHub oder GitLab) ausgewählt werden können.	Partiell Implementiert	In der aktuellen Entwicklungsstufe wurden verstärkt nur die Fälle berücksichtigt, in denen eine Konfiguration per GitHub eingerichtet wurde.
F4.3	Auswahl der Synchronisationsoptionen	Ein Git-Repository, welches Konfigurationen für das Tool bereitstellt, MUSS die Synchronisationsoptionen 'Manuel' oder 'Verwaltet' verfügen. Die Option 'Verwaltet' VERBIETET manuelle Einstellungsmöglichkeiten des Nutzers.	Implementiert	
F5	**Erweiterbarkeit durch den Benutzer**			
F5.0	Bereitstellung von Aktualisierungen	Das Tool MUSS Möglichkeiten Anbieten die es ermöglichen, seine Komponenten zu aktualisieren.	Implementiert	

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
F5.1	Aktualisierung des Tools	Das Tool MUSS Möglichkeiten Anbieten die es ermöglichen, seine Komponenten zu updaten.	Nicht Implementiert	Derzeit ist das Tool nicht im VS Marketplace gelistet. Eine Aktualisierung ist daher nur durch das manuelle Herunterladen und Installieren neuerer Versionen vom GitHub Repository des Tools möglich.
F5.2	Verwalten von Codeanalysen	Das Tool MUSS das Verwalten (Installieren, Aktualisieren, Löschen und Einsehen) von Verfügbaren Codeanalysekomponenten ermöglichen.	Implementiert	Diese Funktionalität ist nur dann verfügbar, wenn ein Konfigurationsrepository angegeben wurde.
F6	**Erweiterbarkeit durch Erweiterungsentwickler**			
F6.0	Entwickeln von Erweiterungen	Das Tool MUSS ein SDK bereitstellen, mit welcher alle Funktionen von F1.0 bis F5.2 erweitert werden können.	Vorhanden	
F6.1	Bereitstellen einer Dokumentation	Das in F6.0 erwähnte SDK MUSS mittels XML-Dokumentationskommentaren[2] dokumentiert sein.	Vorhanden	
	Nichtfunktionale Anforderungen			
NF1	**IDE Integration**			

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
NF1.0	Visual Studio Integration	Das Tool MUSS ein kompatibles Plugin für die IDE Visual Studio 2017 sein. Die Installation des Tools MUSS durch eine geeignete Installationsmethode (.vsix oder .msi) erfolgen.	Implementiert	Ältere und neuere Version sind nicht getestet. VS 2017 ist die derzeit neuste, öffentliche Version.
NF1.1	Keine Unterbrechung von Arbeitsprozessen	Das Tool DARF NICHT laufende Arbeitsprozesse der IDE behindern, ausbremsen oder diese blockieren. Codeanalysen MÜSSEN als Hintergrundprozess laufen.	Implementiert	
NF1.2	Kein Absturz der IDE	Das Tool DARF die IDE NICHT zum Absturz bringen oder einen Datenverlust in Form von Quellcode o.ä. verursachen.	Teilweise Implementiert	Aktuell handelt es sich um ein Produkt im Betastadium. Es kann daher nicht vollends ausgeschlossen werden, dass Abstürze des Tools auftreten.
NF1.3	Installationshinweise	Es MUSS klar erkennbar sein, ob das Tool installiert ist. Hierzu DÜRFEN eigene Menüpunkte o.ä. verwendet werden.	Implementiert	
NF2	**Bedienbarkeit**			

Nr.	Bezeichnung	Beschreibung	Status	Bermerkungen
NF2.0	Benutzbarkeit für Anfänger	Ultimatives Ziel ist, dass das Tool von Entwickler benutzt werden MUSS, die nur wenig Ahnung und Erfahrung mit der Verwendung der .NET Krypto-API haben.	Implementiert	
NF2.1	Bekannte Entwicklungskonzepte	Die API des Tools MUSS Programmierkonzepte enthalten, die allgemein einem C#-Entwickler bekannt sein sollten.	Implementiert	
NF3	**Benutzeroberfläche**			
NF3.0	Verwendung bekannter Interaktionskonzepte	Das Tool MUSS die aus Visual Studio und Windows bekannten Interaktionskomponenten verwenden.	Implementiert	
NF3.1	Natürliche Sprache	Das Tool MUSS die Dialoge und Meldungen in einer für C# Entwickler gebräuchlichen Sprache ausgeben. Fachbegriffe aus dem Bereich der Informationssicherheit SOLLEN erklärt werden.	Partiell Implementiert	
NF3.2	Lokalisierung	Das Tool SOLL in weiteren Sprachen verfügbar sein.	Nicht Implementiert	

[2]https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments, Zugriff: 10.12.2018

Anhang B

Analyzers in Sharper Crypto-API Analysis

Analyzers				
ID	Bezeichnung	Beschreibung	Status	Erkennungs-klasse
SCAA001	Konstanter bzw. hartkodierter IV	Der IV bei einem symmetrischem Verschlüssel-ungsverfahren darf nicht kon-stant oder hartkodiert sein.	Implementiert	Schwache Konfigura-tion der be-nutzten API / Auslesbare Information
SCAA002	Konstanter bzw. hartkodierter Schlüssel	Der Schlüssel bei einem sym-metrischem Verschlüssel-ungsverfahren darf nicht kon-stant oder hartkodiert sein.	Implementiert	Auslesbare Information
SCAA003	IV und Schlüssel sind identisch	IV und Schlüssel dürfen nicht identisch sein. IV und Schlüssel sollen nicht voneinander abgeleitet sein.	Implementiert	Auslesbare Information
SCAA004	Schwacher Zufall	Der verwendete Zufallsgenerator muss sicher ge-gen Erraten und Errechnen des nächsten oder vorheriger Werte sein.	Implementiert	Verstoß ge-gen die API-Spezifikation

ID	Bezeichnung	Beschreibung	Status	Erkennungs-klasse
SCAA005	Hartkodierte Zugangsdaten	Der Quellcode darf keine hartkodierten Zugangsdaten wie Benutzername oder Passwort enthalten.	Implementiert	Auslesbare Information
SCAA006	Code ist nicht FIPS konform	Die API stellt FIPS konforme Funktionen bereit, die verwendet werden können.	Implementiert	Schwache Konfiguration der benutzten API
SCAA007	Schwache Hashingverfahren	Es sollen keine schwachen Hashingverfahren wie MD5 verwendet werden.	Implementiert	Schwache Hashingalgorithmen
SCAA008	Schwache symmetrische Verschlüsselungsverfahren	Es sollen keine schwachen Verschlüsselungsverfahren wie DES verwendet werden.	Implementiert	Schwache Verschlüsselungsalgorithmen
SCAA009	Schwacher Betriebsmodus	Der Betriebsmodus ECB darf nicht für Blockverschlüsselungen verwendet werden.	Implementiert	Schwache Konfiguration der benutzten API
SCAA010	Konstantes bzw. hartkodiertes Salt	Das Salt bei einer passwortbasierten Verschlüsselung darf nicht konstant oder hartkodiert sein.	Implementiert	Schwache Konfiguration der benutzten API
SCAA011	Zu geringer Kostenfaktor	Der Kostenfaktor, bei einer Schlüsselableitungsfunktion muss ausreichend hoch sein.	Implementiert	Schwache Konfiguration der benutzten API

ID	Bezeichnung	Beschreibung	Status	Erkennungs-klasse
SCAA012	Zu geringe Länge des Salt	Die Länge des Salt muss mindestens 16 Bytes lang sein.	Implementiert	Schwache Konfiguration der benutzten API

Anhang C

Inhalt der Begleit-CD

Auf der Begleit-CD sind der gesamte Quellcode von *Sharper Crypto-API Analysis* inklusive Installationsdokumentation, alle in dieser Arbeit verwendetet Bilder sowie die Arbeit selbst im PDF-Format enthalten. Zur Verwendung und Erstellung des Tools wurde ein erklärendes Textdokument auf der Begleit-CD im Markdown-Format (Readme.md) angelegt.

Der Verzeichnisbaum der Begleit-CD ist in Listing C.1 angegeben:

```
Begleit-CD
+---Arbeit
|   \   Masterarbeit_Sebastian_Leuer.pdf
+---Grafiken
+---Repository
|   |   Readme.md
|   |   LICENSE
|   |   SharperCryptoApiAnalysis.sln
|   +---src
|   \---submodules
\   SharperCryptoAPIAnalysis.vsix
```

Listing C.1: Verzeichnisbaum der Begleit-CD